Arrakis: The Operating System Is the Control Plane

SIMON PETER, JIALIN LI, IRENE ZHANG, DAN R. K. PORTS, DOUG WOOS,
ARVIND KRISHNAMURTHY, and THOMAS ANDERSON, University of Washington
TIMOTHY ROSCOE, ETH Zurich

Recent device hardware trends enable a new approach to the design of network server operating systems. In a traditional operating system, the kernel mediates access to device hardware by server applications to enforce process isolation as well as network and disk security. We have designed and implemented a new operating system, Arrakis, that splits the traditional role of the kernel in two. Applications have direct access to virtualized I/O devices, allowing most I/O operations to skip the kernel entirely, while the kernel is re-engineered to provide network and disk protection without kernel mediation of every operation. We describe the hardware and software changes needed to take advantage of this new abstraction, and we illustrate its power by showing improvements of 2 to 5× in latency and 9× throughput for a popular persistent NoSQL store relative to a well-tuned Linux implementation.

CCS Concepts: • **Software and its engineering** → **Operating systems**; **Input/output**; • **Information systems** → *Storage virtualization*; • **Networks** → *Network servers*; *Cloud computing;*

Additional Key Words and Phrases: Kernel bypass, I/O virtualization, SR-IOV

ACM Reference Format:
Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The operating system is the control plane. ACM Trans. Comput. Syst. 33, 4, Article 11 (November 2015), 30 pages.
DOI: http://dx.doi.org/10.1145/2812806

1. INTRODUCTION

Reducing the overhead of the operating system process abstraction has been a long-standing goal of systems design. This issue has become particularly salient with modern client-server computing. The combination of high-speed Ethernet and low-latency persistent memories is considerably raising the efficiency bar for I/O-intensive software. Many servers spend much of their time executing operating system code: delivering interrupts, demultiplexing and copying network packets, and maintaining file system metadata. Server applications often perform very simple functions, such as key-value table lookup and storage, yet traverse the OS kernel multiple times per client request.

These trends have led to a long line of research aimed at optimizing kernel code paths for various use cases: eliminating redundant copies in the kernel [Pai et al. 1999], reducing the overhead for large numbers of connections [Han et al. 2012], protocol specialization [Mosberger and Peterson 1996], resource containers [Leslie et al. 1996; Banga et al.

This work was supported by NetApp, Google, and the National Science Foundation.
This work is based on a prior publication in OSDI 2014.
Authors' addresses: S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, and T. Anderson, University of Washington, Computer Science & Engineering, Box 352350, Seattle, WA 98195-2350, USA; emails: {simpeter, lijl, iyzhang, drkp, dwoos, arvind, tom}@cs.washington.edu; T. Roscoe, ETH Zurich, Systems Group Administration, CAB F 78, Universitaetstrasse 6, 8092 Zürich; email: troscoe@inf.ethz.ch.

ACM 0734-2071/2015/11-ART11 $15.00
DOI: http://dx.doi.org/10.1145/2812806

1999], direct transfers between disk and network buffers [Pai et al. 1999], interrupt steering [Pesterev et al. 2012], system call batching [Rizzo 2012], hardware TCP acceleration, and so forth. Much of this has been adopted in mainline commercial OSs, yet it has been a losing battle: we show that the Linux network and file system stacks have latency and throughput many times worse than that achieved by the raw hardware.

Twenty years ago, researchers proposed streamlining packet handling for parallel computing over a network of workstations by mapping the network hardware directly into user space [von Eicken et al. 1995; Compaq Computer Corp. et al. 1997; Druschel et al. 1994]. Although commercially unsuccessful at the time, the virtualization market has now led hardware vendors to revive the idea [Abramson 2006; Kutch 2011; Consortium 2009] and also extend it to disks [Trivedi et al. 2013; Volos et al. 2014].

This article explores the implications of removing the kernel from the data path for nearly all I/O operations in the context of a general-purpose server OS, rather than a virtual machine monitor. We argue that doing this must provide applications with the same security model as traditional designs; it is easy to get good performance by extending the trusted computing base to include application code, for example, by allowing applications unfiltered direct access to the network/disk.

We demonstrate that operating system protection is not contradictory with high performance. For our prototype implementation, a client request to the Redis persistent NoSQL store has 2× better read latency, 5× better write latency, and 9× better write throughput compared to Linux.

We make three specific contributions:

- We give an architecture for the division of labor between the device hardware, kernel, and runtime for direct network and disk I/O by unprivileged processes, and we show how to efficiently emulate our model for I/O devices that do not fully support virtualization (Section 3).
- We implement a prototype of our model as a set of modifications to the open-source Barrelfish[1] operating system, running on commercially available multicore computers and I/O device hardware (Section 3.8).
- We use our prototype to quantify the potential benefits of user-level I/O for several widely used network services, including a distributed object cache, Redis; an IP-layer middlebox; and an HTTP load balancer (Section 4). We show that significant gains are possible in terms of both latency and scalability, relative to Linux, in many cases without modifying the application programming interface; additional gains are possible by changing the POSIX API (Section 4.3).

2. BACKGROUND

We first give a detailed breakdown of the OS and application overheads in network and storage operations today, followed by a discussion of current hardware technologies that support user-level networking and I/O virtualization.

To analyze the sources of overhead, we record timestamps at various stages of kernel and user-space processing. Our experiments are conducted on a six-machine cluster consisting of six-core Intel Xeon E5-2430 (Sandy Bridge) systems at 2.2GHz running Ubuntu Linux 13.04. The systems have an Intel X520 (82599-based) 10Gb Ethernet adapter [Intel Corporation 2010] and an Intel MegaRAID RS3DC040 RAID controller [Intel Corporation 2013b] with 1GB of flash-backed DRAM attached to a 100GB Intel DC S3700 SSD. All machines are connected to a 10Gb Dell PowerConnect 8024F Ethernet switch. One system (the *server*) executes the application under scrutiny, while the others act as clients.

[1]Available from http://www.barrelfish.org/. All URLs last accessed on August 26, 2015.

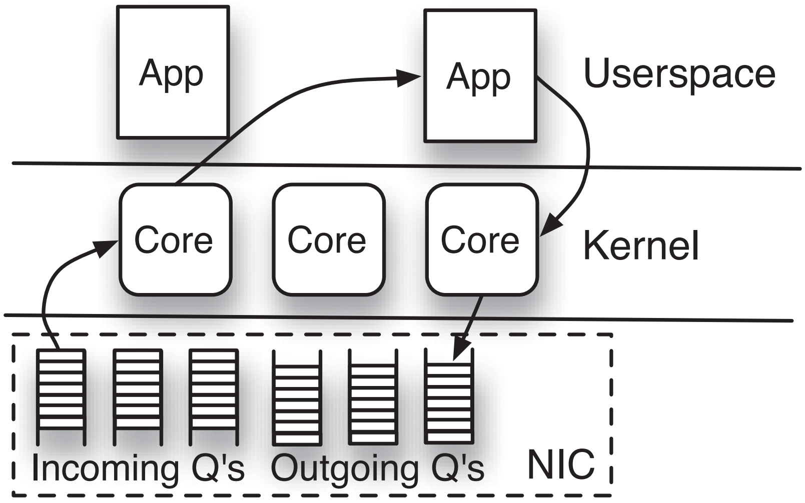

Fig. 1. Linux networking architecture and workflow.

Table I. Sources of Packet Processing Overhead in Linux and Arrakis
All times are averages over 1,000 samples, given in μs (and standard deviation for totals). Arrakis/P uses the POSIX interface; Arrakis/N uses the native Arrakis Interface.

		Linux				Arrakis			
		Receiver running		CPU idle		Arrakis/P		Arrakis/N	
Network stack	in	1.26	(37.6%)	1.24	(20.0%)	0.32	(22.3%)	0.21	(55.3%)
	out	1.05	(31.3%)	1.42	(22.9%)	0.27	(18.7%)	0.17	(44.7%)
Scheduler		0.17	(5.0%)	2.40	(38.8%)	-		-	
Copy	in	0.24	(7.1%)	0.25	(4.0%)	0.27	(18.7%)	-	
	out	0.44	(13.2%)	0.55	(8.9%)	0.58	(40.3%)	-	
Kernel crossing	return	0.10	(2.9%)	0.20	(3.3%)	-		-	
	syscall	0.10	(2.9%)	0.13	(2.1%)	-		-	
Total		3.36		6.19		1.44		0.38	
Std. dev.		0.66		0.82		<0.01		<0.01	

The combination of RAID controller and single SSD is the highest-performing configuration for this hardware. We use this setup in our experiments to represent recent high-performance storage devices, such as NVMe SSDs [Intel Corporation 2013d].

2.1. Networking Stack Overheads

Consider a UDP echo server implemented as a Linux process. The server performs **recvmsg** and **sendmsg** calls in a loop, with no application-level processing, so it stresses packet processing in the OS. Figure 1 depicts the typical workflow for such an application. As Table I shows, operating system overhead for packet processing falls into four main categories:

- **Network stack costs**: packet processing at the hardware, IP, and UDP layers
- **Scheduler overhead**: waking up a process (if necessary), selecting it to run, and context switching to it
- **Kernel crossings**: from kernel to user space and back
- **Copying of packet data**: from the kernel to a user buffer on receive, and back on send

Of the total 3.36μs (see Table I) spent processing each packet in Linux, nearly 70% is spent in the network stack. This work is mostly software demultiplexing, security checks, and overhead due to indirection at various layers. The kernel must validate the header of incoming packets and perform security checks on arguments provided by the application when it sends a packet. The stack also performs checks at layer boundaries.

Scheduler overhead depends significantly on whether the receiving process is currently running. If it is, only 5% of processing time is spent in the scheduler; if it is not, the time to context-switch to the server process from the idle process adds an extra 2.2μs and a further 0.6μs slowdown in other parts of the network stack.

Cache and lock contention issues on multicore systems add further overhead and are exacerbated by the fact that incoming messages can be delivered on different queues by the network card, causing them to be processed by different CPU cores—which may not be the same as the cores on which the user-level process is scheduled, as depicted in Figure 1. Advanced hardware support such as accelerated receive flow steering[2] aims to mitigate this cost, but these solutions themselves impose nontrivial setup costs [Pesterev et al. 2012].

By leveraging hardware support to remove kernel mediation from the data plane, Arrakis can eliminate certain categories of overhead entirely, and minimize the effect of others. Table I also shows the corresponding overhead for two variants of Arrakis. Arrakis eliminates scheduling and kernel crossing overhead entirely, because packets are delivered directly to user space. Network stack processing is still required, of course, but it is greatly simplified: it is no longer necessary to demultiplex packets for different applications, and the user-level network stack need not validate parameters provided by the user as extensively as a kernel implementation must. Because each application has a separate network stack, and packets are delivered to cores where the application is running, lock contention and cache effects are reduced.

In the Arrakis network stack, the time to copy packet data to and from user-provided buffers dominates the processing cost, a consequence of the mismatch between the POSIX interface (Arrakis /P) and NIC packet queues. Arriving data is first placed by the network hardware into a network buffer and then copied into the location specified by the POSIX read call. Data to be transmitted is moved into a buffer that can be placed in the network hardware queue; the POSIX write can then return, allowing the user memory to be reused before the data is sent. Although researchers have investigated ways to eliminate this copy from kernel network stacks [Pai et al. 1999], as Table I shows, most of the overhead for a kernel-resident network stack is elsewhere. Once the overhead of traversing the kernel is removed, there is an opportunity to rethink the POSIX API for more streamlined networking. In addition to a POSIX-compatible interface, Arrakis provides a native interface (Arrakis/N), which supports true zero-copy I/O.

2.2. Storage Stack Overheads

To illustrate the overhead of today's OS storage stacks, we conduct an experiment, where we execute small write operations immediately followed by an fsync[3] system call in a tight loop of 10,000 iterations, measuring each operation's latency. We store the file system on a RAM disk, so the measured latencies represent purely CPU overhead.

The overheads shown in Figure 2 stem from data copying between user and kernel space, parameter and access control checks, block and inode allocation, virtualization (the VFS layer), snapshot maintenance (btrfs), and metadata updates, in many cases

[2]https://www.kernel.org/doc/Documentation/networking/scaling.txt.
[3]We also tried fdatasync, with negligible difference in latency.

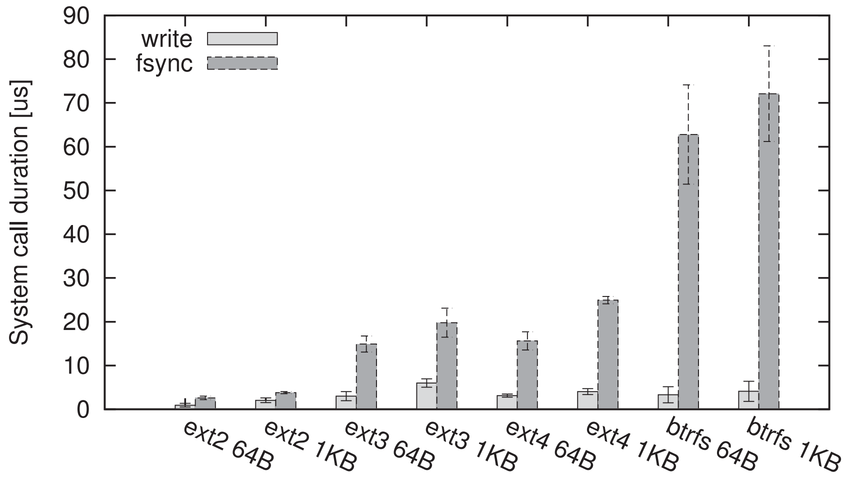

Fig. 2. Average overhead in μs of various Linux file system implementations, when conducting small, persistent writes. Error bars show standard deviation.

via a journal [Volos et al. 2014]. They match the overheads previously reported for these file systems [Volos et al. 2014; Caulfield et al. 2012].

While historically these CPU overheads have been insignificant compared to disk access time, recent hardware trends have drastically reduced common-case write storage latency by introducing flash-backed DRAM onto the device. In these systems, OS storage stack overhead becomes a major factor. We measured average write latency to our RAID cache to be 25μs. PCIe-attached flash storage adapters, like Fusion-IO's ioDrive2, report hardware access latencies as low as 15μs [Fusion-IO 2014]. In comparison, OS storage stack overheads are high, adding between 40% (ext2) and 200% (ext4) for the Linux extended file system family, and up to 5× for btrfs. The large standard deviation for btrfs stems from its highly threaded design, used to flush noncritical file system metadata and update reference counts in the background.

2.3. Application Overheads

What do these I/O stack overheads mean to operation latencies within a typical data center application? Consider the Redis[4] NoSQL store. Redis persists each write via an operational log (called *append-only file*)[5] and serves reads from an in-memory data structure.

To serve a read, Redis performs a series of operations: First, `epoll` is called to await data for reading, followed by `recv` to receive a request. After receiving, the (textual) request is parsed and the key looked up in memory. Once found, a response is prepared and then, after `epoll` is called again to check whether the socket is ready, it is sent to the client via `send`. For writes, Redis additionally marshals the operation into log format, `writes` the log, and waits for persistence (via the `fsync` call) before responding. Redis also spends time in accounting, access checks, and connection handling (**Other** row in Table II).

Table II shows that a total of 76% of the latency in an average read hit on Linux is due to socket operations. In Arrakis, we reduce socket operation latency by 68%.

[4]http://redis.io/.
[5]Redis also supports snapshot persistence because of the high per-operation overhead imposed by Linux.

Table II. Overheads in the Redis NoSQL Store for Memory Reads (Hits) and Durable Writes (Legend in Table I)

	Read Hit				Durable Write			
	Linux		Arrakis/P		Linux		Arrakis/P	
`epoll`	2.42	(27.91%)	1.12	(27.52%)	2.64	(1.62%)	1.49	(4.73%)
`recv`	0.98	(11.30%)	0.29	(7.13%)	1.55	(0.95%)	0.66	(2.09%)
Parse input	0.85	(9.80%)	0.66	(16.22%)	2.34	(1.43%)	1.19	(3.78%)
Find/set key	0.10	(1.15%)	0.10	(2.46%)	1.03	(0.63%)	0.43	(1.36%)
Log marshal	-		-		3.64	(2.23%)	2.43	(7.71%)
`write`	-		-		6.33	(3.88%)	0.10	(0.32%)
`fsync`	-		-		137.84	(84.49%)	24.26	(76.99%)
Prep response	0.60	(6.92%)	0.64	(15.72%)	0.59	(0.36%)	0.10	(0.32%)
`send`	3.17	(36.56%)	0.71	(17.44%)	5.06	(3.10%)	0.33	(1.05%)
Other	0.55	(6.34%)	0.46	(11.30%)	2.12	(1.30%)	0.52	(1.65%)
Total	8.67	($\sigma = 2.55$)	4.07	($\sigma = 0.44$)	163.14	($\sigma = 13.68$)	31.51	($\sigma = 1.91$)
99th percentile	15.21		4.25		188.67		35.76	

Similarly, 90% of the latency of a write on Linux is due to I/O operations. In Arrakis, we reduce I/O latency by 82%.

We can also see that Arrakis reduces some application-level overheads. This reduction is due to better cache behavior of the user-level I/O stacks and the control/data plane separation avoiding all kernel crossings. Arrakis's write latency is still dominated by storage access latency (25μs in our system). We expect the gap between Linux and Arrakis performance to widen as faster storage devices appear on the market.

It is tempting to think that zero-copy I/O could be provided in a conventional OS like Linux simply by modifying its interface in the same way. However, eliminating copying entirely is possible only because Arrakis eliminates kernel crossings and kernel packet demultiplexing as well. Using hardware demultiplexing, Arrakis can deliver packets directly to a user-provided buffer, which would not be possible in Linux because the kernel must first read and process the packet to determine which user process to deliver it to. On the transmit side, the application must be notified once the send operation has completed and the buffer is available for reuse; this notification would ordinarily require a kernel crossing.

2.4. Tail Latency

In addition to the average latency of operations, the *latency tail* plays a significant role for many large-scale data center application deployments. Large-scale deployments typically involve many interactions among back-end services to serve one front-end user request. For example, hundreds of key-value operations might be executed on a variety of servers to service one user request. When the request must wait for the last of these operations to complete, the *slowest* operation determines the latency of the system. As a result, data center application developers typically aim to optimize the 99th (or 99.9th) percentile latency Dean and Barroso [2013].

Operating system I/O stacks are a major component of application tail behavior [Li et al. 2014]. Shared code paths, interrupt scheduling, and contention on shared data can prolong request processing arbitrarily. Figure 3 demonstrates that Redis operation latency on Linux varies much more than that on Arrakis. To show this, we use a complementary cumulative distribution function (CCDF), plotting the probability that an operation takes at least as long as the value shown on the x-axis. We also repeat the 99th percentile latencies in Table II. Arrakis obeys very tight latency bounds for both read and write operations. The 99th percentile latency of reads is well within 1μs of the average and that of writes is within 5μs. A few write operations take considerably longer in Arrakis. This is due to the behavior of the RAID controller, which, in some cases, deviates from its average hardware write latency of 25μs.

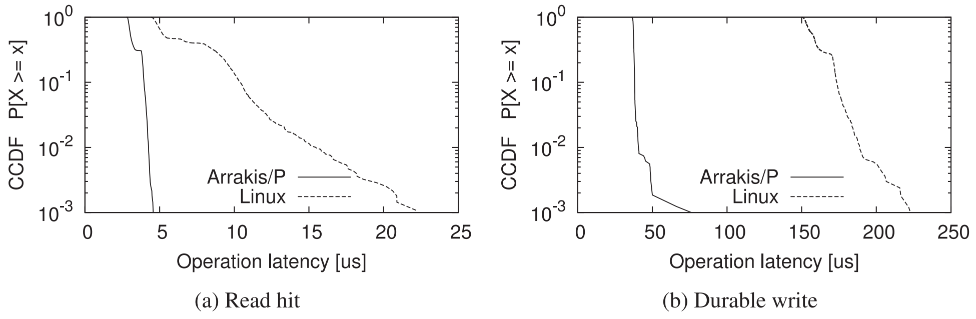

Fig. 3. Latency distribution of Redis read and write operations.

2.5. Hardware I/O Virtualization

Single-Root I/O Virtualization (SR-IOV) [Kutch 2011] is a hardware technology intended to support high-speed I/O for multiple virtual machines sharing a single physical machine. An SR-IOV-capable I/O adapter appears on the PCIe interconnect as a single "physical function" (PCI parlance for a device), which can in turn dynamically create additional "virtual functions." Each of these virtual functions appears as a different PCI device that can be directly mapped in a different virtual machine, and access to which can be protected via an IOMMU (e.g., as in Intel's VT-d architecture [Intel Corporation 2013c]). To the guest operating system, each virtual function can be programmed as if it were a regular physical device, with a normal device driver and an unmodified I/O stack. Hypervisor software with access to the physical hardware (such as Domain 0 in a Xen [Barham et al. 2003] installation) creates and deletes these virtual functions, and configures filters in the SR-IOV adapter to demultiplex hardware operations to different virtual functions and therefore different guest operating systems.

SR-IOV is intended to reduce the overhead of OS virtualization. In conjunction with platform hardware support for virtualization (such as IOMMUs and direct delivery of interrupts to virtual machines), it can effectively remove the hypervisor from the data path. However, when used as intended, the guest OS still mediates access to data packets.

In Arrakis, we use SR-IOV, the IOMMU, and supporting adapters to provide direct application-level access to I/O devices. This is a modern implementation of an idea that was implemented 20 years ago with U-Net [von Eicken et al. 1995] but generalized to flash storage and Ethernet network adapters. To make user-level I/O stacks tractable, we need a hardware-independent device model and API that capture the important features of SR-IOV adapters [Solarflare Communications, Inc. 2010; Intel Corporation 2010; LSI Corporation 2010, 2014]; a hardware-specific device driver matches our API to the specifics of the particular device. We discuss this model in the next section, along with potential improvements to the existing hardware to better support user-level I/O.

Remote Direct Memory Access (RDMA) is another popular model for user-level networking [Consortium 2009]. RDMA gives applications the ability to read from or write to a region of virtual memory on a remote machine directly from user-space, bypassing the operating system kernel on both sides. The intended use case is for a parallel program to be able to directly read and modify its data structures even when they are stored on remote machines.

Most importantly, RDMA is point to point. Each participant receives an authenticator providing it permission to remotely read/write a particular region of memory. Since clients in client-server computing are not mutually trusted, the hardware would need to keep a separate region of memory for each active connection. This makes it

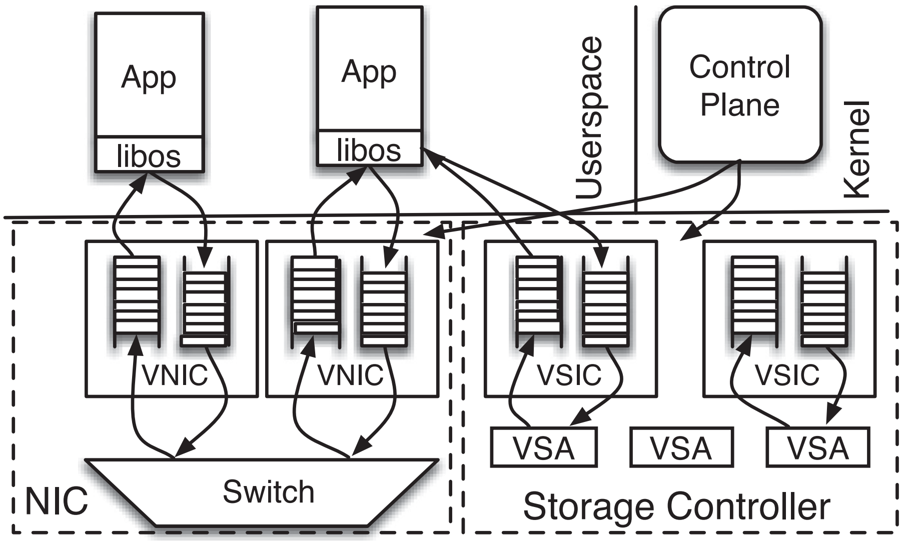

Fig. 4. Arrakis architecture. The storage controller maps VSAs to physical storage.

challenging to apply the model to a broader class of client-server applications. For example, FaRM [Dragojević et al. 2014] describes the various complications that have to be taken into account when implementing a simple message-passing queue via RDMA, such as bad client scalability due to polling overheads and translation/authenticator cache overruns on the NIC. Therefore, we do not consider RDMA operations here.

3. DESIGN AND IMPLEMENTATION

Arrakis has the following design goals:

- **Minimize kernel involvement for data-plane operations:** Arrakis is designed to limit or remove kernel mediation for most I/O operations. I/O requests are routed to and from the application's address space without requiring kernel involvement and without sacrificing security and isolation properties.
- **Transparency to the application programmer:** Arrakis is designed to significantly improve performance without requiring modifications to applications written to the POSIX API. Additional performance gains are possible if the developer can modify the application.
- **Appropriate OS/hardware abstractions:** Arrakis's abstractions should be sufficiently flexible to efficiently support a broad range of I/O patterns, scale well on multicore systems, and support application requirements for locality and load balance.

In this section, we show how we achieve these goals in Arrakis. We describe an ideal set of hardware facilities that should be present to take full advantage of this architecture, and we detail the design of the control plane and data plane interfaces that we provide to the application. Finally, we describe our implementation of Arrakis based on the Barrelfish operating system.

3.1. Architecture Overview

Arrakis targets I/O hardware with support for virtualization, and Figure 4 shows the overall architecture. In this article, we focus on hardware that can present multiple instances of itself to the operating system and the applications running on the node. For each of these virtualized device instances, the underlying physical device provides unique memory mapped register files, descriptor queues, and interrupts, hence allowing the control plane to map each device instance to a separate protection domain. The device exports a management interface that is accessible from the control plane in order to create or destroy virtual device instances, associate individual instances with network flows or storage areas, and allocate shared resources to the different instances. Applications conduct I/O through their protected virtual device instance without requiring kernel intervention. In order to perform these operations, applications rely on a user-level I/O stack that is provided as a library. The user-level I/O stack can be tailored to the application as it can assume exclusive access to a virtualized device instance, allowing us to remove any features not necessary for the application's functionality. Finally, (de)multiplexing operations and security checks are not needed in this dedicated environment and can be removed.

The user naming and protection model is unmodified. A global naming system is provided by the control plane. This is especially important for sharing stored data. Applications implement their own storage, while the control plane manages naming and coarse-grained allocation, by associating each application with the directories and files it manages. Other applications can still read those files indirectly through the kernel, which hands the directory or read request to the appropriate application.

3.2. Hardware Model

A key element of our work is to develop a hardware-independent layer for virtualized I/O—that is, a device model providing an "ideal" set of hardware features. This device model captures the functionality required to implement in hardware the data plane operations of a traditional kernel. Our model resembles what is already provided by some hardware I/O adapters; we hope it will provide guidance as to what is needed to support secure user-level networking and storage.

In particular, we assume our network devices provide support for virtualization by presenting themselves as multiple *virtual network interface cards (VNICs)* and that they can also multiplex/demultiplex packets based on complex filter expressions, directly to queues that can be managed entirely in user space without the need for kernel intervention. Similarly, each storage controller exposes multiple *virtual storage interface controllers* (VSICs) in our model. Each VSIC provides independent storage command queues (e.g., of SCSI or ATA format) that are multiplexed by the hardware. Associated with each such virtual interface card (VIC) are *queues* and *rate limiters*. VNICs also provide *filters* and VSICs provide *virtual storage areas*. We discuss these components next.

Queues: Each VIC contains multiple pairs of DMA queues for user-space send and receive. The exact form of these VIC queues could depend on the specifics of the I/O interface card. For example, it could support a scatter/gather interface to aggregate multiple physically disjoint memory regions into a single data transfer. For NICs, it could also optionally support hardware checksum offload and TCP segmentation facilities. These features enable I/O to be handled more efficiently by performing additional work in hardware. In such cases, the Arrakis system offloads operations and further reduces overheads.

Transmit and receive filters: A transmit filter is a predicate on network packet header fields that the hardware will use to determine whether to send the packet or discard it (possibly signaling an error either to the application or the OS). The transmit filter prevents applications from spoofing information such as IP addresses and VLAN tags and thus eliminates kernel mediation to enforce these security checks. It can also be used to limit an application to communicate with only a preselected set of nodes.

A receive filter is a similar predicate that determines which packets received from the network will be delivered to a VNIC and to a specific queue associated with the target VNIC. For example, a VNIC can be set up to receive all packets sent to a particular port, so both connection setup and data transfers can happen at the user level. Installation of transmit and receive filters are privileged operations performed via the kernel control plane.

Virtual storage areas: Storage controllers need to provide an interface via their physical function to map *virtual storage areas* (VSAs) to extents of physical drives and associate them with VSICs. A typical VSA will be large enough to allow the application to ignore the underlying multiplexing—for example, multiple erasure blocks on flash, or cylinder groups on disk. An application can store multiple subdirectories and files in a single VSA, providing precise control over multiobject serialization constraints.

A VSA is thus a persistent segment [Bensoussan et al. 1972]. Applications reference blocks in the VSA using virtual offsets, converted by hardware into physical storage locations. A VSIC may have multiple VSAs, and each VSA may be mapped into multiple VSICs for interprocess sharing.

Bandwidth allocators: This includes support for resource allocation mechanisms such as rate limiters and pacing/traffic shaping of I/O. Once a frame has been removed from a transmit rate-limited or paced queue, the next time another frame could be fetched from that queue is regulated by the rate limits and the interpacket pacing controls associated with the queue. Installation of these controls is also a privileged operation.

In addition, we assume that the I/O device driver supports an introspection interface, allowing the control plane to query for resource limits (e.g., the number of queues) and check for the availability of hardware support for I/O processing (e.g., checksumming or segmentation).

Network cards that support SR-IOV have the key elements of this model: they allow the creation of multiple VNICs that each may have multiple send and receive queues, and they support at least rudimentary transmit and receive filters. Not all NICs provide the rich filtering semantics we desire; for example, the Intel 82599 can filter only based on source or destination MAC addresses and VLAN tags, not arbitrary predicates on header fields. However, this capability is within reach: some network cards (e.g., Solarflare 10Gb adapters) can already filter packets on all header fields, and the hardware support required for more general VNIC transmit and receive filtering is closely related to that used for techniques like Receive-Side Scaling, which is ubiquitous in high-performance network cards.

Storage controllers have some parts of the technology needed to provide the interface we describe. For example, RAID adapters have a translation layer that is able to provide virtual disks above physical extents, and SSDs use a flash translation layer for wear leveling. SCSI host-bus adapters support SR-IOV technology for virtualization [LSI Corporation 2010, 2014] and can expose multiple VSICs, and the NVMe standard proposes multiple command queues for scalability [Intel Corporation 2013d]. Only the required protection mechanism is missing. We anticipate VSAs to be allocated in large chunks and thus hardware protection mechanisms can be coarse-grained and lightweight.

Although some proposals for NVRAM technology have called for storage class memories to be incorporated into the memory interconnect, we anticipate that DMA-based protocols will remain available, given the need to provide compatibility with existing devices, as well as to provide an asynchronous interface to devices with higher latency than DRAM.

Finally, the number of hardware-supported VICs might be limited. The 82599 [Intel Corporation 2010] and SAS3008 [LSI Corporation 2014] support 64. This number is adequate with respect to the capabilities of the rest of the hardware (e.g., the number of CPU cores), but we expect it to rise. The PCI working group has already ratified an addendum to SR-IOV that increases the supported number of virtual functions to 2,048. Bandwidth allocation within the 82599 is limited to weighted round-robin scheduling and rate limiting of each of the 128 transmit/receive queues. Recent research has demonstrated that precise rate limiting in hardware can scale to tens of thousands of traffic classes, enabling sophisticated bandwidth allocation policies [Radhakrishnan et al. 2014].

Arrakis currently assumes hardware that can filter and demultiplex flows at a level (packet headers, etc.) corresponding roughly to a traditional OS API, but no higher. An open question is the extent to which hardware that can filter on application-level properties (including content) would provide additional performance benefits.

3.3. VSIC Emulation

To validate our model given limited support from storage devices, we developed prototype VSIC support by dedicating a processor core to emulate the functionality we expect from hardware. The same technique can be used to run Arrakis on systems without VNIC support.

To handle I/O requests from the OS, our RAID controller provides one request and one response descriptor queue of fixed size, implemented as circular buffers along with a software-controlled register (PR) pointing to the head of the request descriptor queue. Request descriptors (RQDs) have a size of 256 bytes and contain an SCSI command, a scatter-gather array of system memory ranges, and a target logical disk number. The SCSI command specifies the type of operation (read or write), total transfer size, and on-disk base logical block address (LBA). The scatter-gather array specifies the request's corresponding regions in system memory. Response descriptors refer to completed RQDs by their queue entry and contain a completion code. An RQD can be reused only after its response is received.

We replicate this setup for each VSIC by allocating queue pairs and register files of the same format in system memory mapped into applications and to the dedicated VSIC core. Like the 82599, we limit the maximum number of VSICs to 64. In addition, the VSIC core keeps an array of up to four VSA mappings for each VSIC that is programmable only from the control plane. The mappings contain the size of the VSA and an LBA offset within a logical disk, effectively specifying an extent.

In the steady state, the VSIC core polls each VSIC's PR and the latest entry of the response queue of the physical controller in a round-robin fashion. When a new RQD is posted via PR_i on VSIC i, the VSIC core interprets the RQD's logical disk number n as a VSA mapping entry and checks whether the corresponding transfer fits within that VSA's boundaries (i.e., $\text{RQD.LBA} + \text{RQD.size} \leq \text{VSA}_n\text{.size}$). If so, the core copies the RQD to the physical controller's queue, adding VSA_n.offset to RQD.LBA, and sets an unused RQD field to identify the corresponding RQD in the source VSIC before updating the controller's PR register. Upon a response from the controller, the VSIC core copies the response to the corresponding VSIC response queue.

Figure 5 illustrates the request part of this setup for a single example application (App 0) using VSIC 0. For VSIC 0, only VSA 0 is assigned to the RAID controller's physical disk 1 at an LBA of 10 billion and a size of 1GB. When App 0 submits a new virtual RQD to VSA 0 at an LBA offset of 100 and a size of 10MB, the VSIC core checks

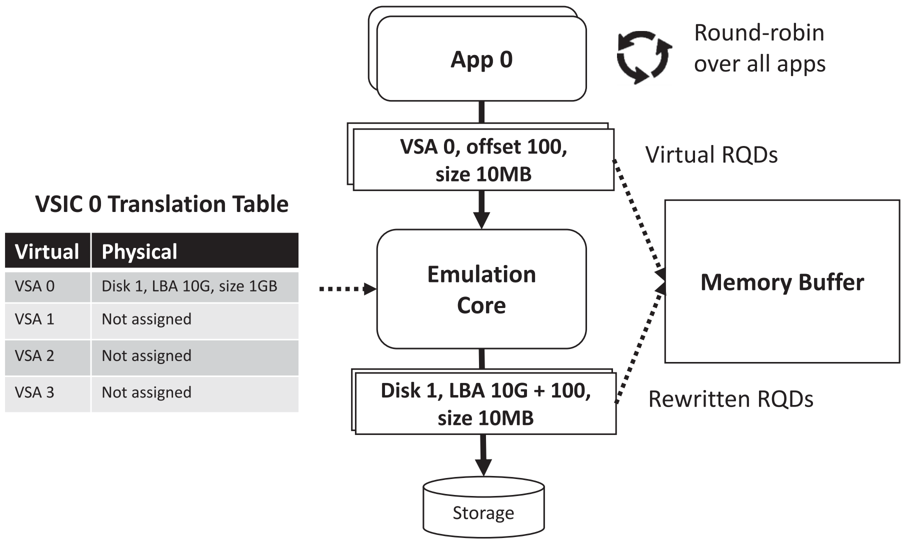

Fig. 5. Arrakis VSIC emulation example for a single application (App 0).

that the RQD specifies an assigned VSA and that the request fits within that VSA. As this is the case, the VSIC core rewrites the RQD according to its internal translation table and the formula given earlier and places it into the RAID controller's descriptor queue. Both virtual and rewritten RQDs refer to the same memory buffer, which is not touched by the VSIC core.

We did not consider VSIC interrupts in our prototype. They can be supported via interprocessor interrupts. To support untrusted applications, our prototype has to translate virtual addresses. This requires it to traverse application page tables for each entry in an RQD's scatter-gather array. In a real system, the IOMMU carries out this task.

On a microbenchmark of 10,000 fixed-size write operations of 1KB via a single VSIC to a single VSA, the average overhead of the emulation is 3μs. Executing virtualization code takes 1μs on the VSIC core; the other 2μs are due to cache overheads that we did not quantify further. To measure the expected VSIC performance with direct hardware support, we map the single RAID hardware VSIC directly into the application memory; we report those results in Section 4.

3.4. Control Plane Interface

The interface between an application and the Arrakis control plane is used to request resources from the system and direct I/O flows to and from user programs. The key abstractions presented by this interface are VICs, doorbells, filters, VSAs, and rate specifiers.

An application can create and delete VICs and associate *doorbells* with particular events on particular VICs. A doorbell is an IPC endpoint used to notify the application that an event (e.g., packet arrival or I/O completion) has occurred and is discussed later. VICs are hardware resources and so Arrakis must allocate them among applications according to an OS policy. Currently this is done on a first-come-first-served basis, followed by spilling to software emulation (Section 3.3).

Filters have a *type* (transmit or receive) and a *predicate,* which corresponds to a convex subvolume of the packet header space (e.g., obtained with a set of mask-and-compare operations). Filters can be used to specify ranges of IP addresses and port numbers associated with valid packets transmitted/received at each VNIC. Filters are a better abstraction for our purposes than a conventional connection identifier (such as a TCP/IP 5-tuple), since they can encode a wider variety of communication patterns, as well as subsume traditional port allocation and interface specification.

For example, in the “map” phase of a MapReduce job, we would like the application to send to, and receive from, an entire class of machines using the same communication endpoint, but nevertheless isolate the data containing the shuffle from other data. As a second example, web servers with a high rate of incoming TCP connections can run into scalability problems processing connection requests [Pesterev et al. 2012]. In Arrakis, a single filter can safely express both a listening socket and all subsequent connections to that socket, allowing server-side TCP connection establishment to avoid kernel mediation.

Applications create a filter with a control plane operation. In the common case, a simple higher-level wrapper suffices: **filter = create_filter(flags, peerlist, servicelist)**. **flags** specifies the filter direction (transmit or receive) and whether the filter refers to the Ethernet, IP, TCP, or UDP header. **peerlist** is a list of accepted communication peers specified according to the filter type, and **servicelist** contains a list of accepted service addresses (e.g., port numbers) for the filter. Wildcards are permitted.

The call to **create_filter** returns **filter**, a kernel-protected capability conferring authority to send or receive packets matching its predicate, and which can then be **assign**ed to a specific queue on a VNIC. VSAs are acquired and assigned to VSICs in a similar fashion.

Similarly, a VSA is acquired via an **acquire_vsa(name) = VSA** call. The returned **VSA** is a capability that allows applications access to that VSA. The control plane stores capabilities and their associations in a separate storage area. If a nonexistent VSA is acquired, its size is initially zero, but it can be resized via the **resize_vsa(VSA, size)** call. VSAs store the mapping of virtual storage blocks to physical ones and are given a global **name**, such that they can be found again when an application is restarted. A VSA can be **assign**ed to a VSIC by the application if it has the appropriate access rights. The control plane then installs the VSA’s mapping within the VSIC, so the application can access its contents.

As long as physical storage space is available, the interface allows creating new VSAs and extending existing ones. Deleting and resizing an existing VSA is always possible. Shrink and delete operations permanently delete the data contained in the VSA by overwriting with zeroes, which can be done in the background by the control plane. Resizing might be done in units of a fixed size if more efficient given the hardware, for example, at the cylinder group level to maintain hard disk head locality. Assuming the storage hardware allows mappings to be made at a block or extent granularity, a regular physical memory allocation algorithm can be used without having to worry about fragmentation. As we envision VSAs to be modified infrequently and by large amounts, the overhead of these operations does not significantly affect application performance.

Finally, a rate specifier can also be assigned to a queue, either to throttle incoming traffic (in the network receive case) or pace outgoing packets and I/O requests. Rate specifiers and filters associated with a VIC queue can be updated dynamically, but all such updates require mediation from the Arrakis control plane.

Our network filters are less expressive than OpenFlow matching tables, in that they do not support priority-based overlapping matches. This is a deliberate choice based on hardware capabilities: NICs today only support simple matching, and to support priorities in the API would lead to unpredictable consumption of hardware resources below the abstraction. Our philosophy is therefore to support expressing such policies only when the hardware can implement them efficiently.

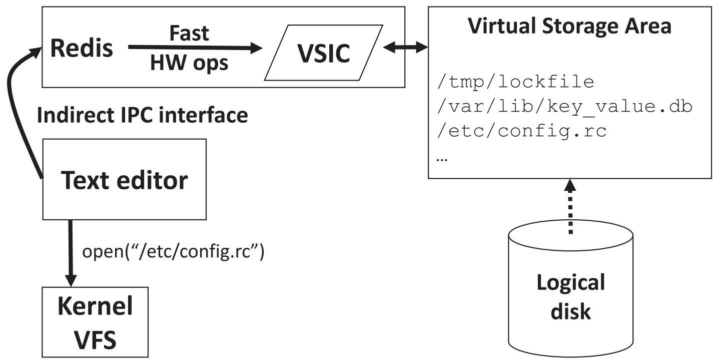

Fig. 6. Arrakis default file access example.

3.5. File Name Lookup

A design principle in Arrakis is to separate file naming from implementation. In a traditional system, the fully qualified filename specifies the file system used to store the file and thus its metadata format. To work around this, many applications build their own metadata indirection inside the file abstraction [Harter et al. 2011]. Instead, Arrakis provides applications direct control over VSA storage allocation: an application is free to use its VSA to store metadata, directories, and file data. To allow other applications access to its data, an application can export file and directory names to the kernel virtual file system (VFS). To the rest of the VFS, an application-managed file or directory appears like a remote mount point—an indirection to a file system implemented elsewhere. Operations within the file or directory are handled locally, without kernel intervention.

Other applications can gain access to these files in three ways. By default, the Arrakis application library managing the VSA exports a file server interface; other applications can use normal POSIX API calls via user-level RPC to the embedded library file server. This library can also run as a standalone process to provide access when the original application is not active. Just like a regular mounted file system, the library needs to implement only functionality required for file access on its VSA and may choose to skip any POSIX features that it does not directly support.

Figure 6 illustrates an example of this default way. In the example, the Redis NoSQL store uses direct, application-level hardware operations on a VSIC mapped into its virtual address space to access data on its own VSA. The VSA is provided by one of the logical disks of the RAID controller. Within its VSA, Redis stores its local data, such as its database and configuration. This makes sense because Redis can gain performance benefits from directly accessing its own data. To allow other applications access to this data, Redis exports appropriate filenames to the kernel-level VFS. This allows the text editor in the example to open the configuration file. It does so by issuing an open system call to the kernel specifying the fully qualified name of Redis's configuration file. The kernel returns a file descriptor that redirects the text editor to make all file access via the POSIX interface of Redis's application library that has direct accesses to the VSA.

A second way to give multiple applications access to a set of files is to map their VSAs into the applications' corresponding processes. If an application, like a virus checker or backup system, has both permission to read the application's metadata and the appropriate library support, it can directly access the file data in the VSA. In this case, access control is done for the entire VSA and not per file or directory. Finally, the user can direct the originating application to export its data into a standard format, such as a PDF file, stored as a normal file in the kernel-provided file system.

The combination of VFS and library code implements POSIX semantics seamlessly. For example, if execute rights are revoked from a directory, the VFS prevents future traversal of that directory's subtree, but existing RPC connections to parts of the subtree may remain intact until closed. This is akin to a POSIX process retaining a subdirectory as the current working directory—relative traversals are still permitted.

3.6. Network Data Plane Interface

In Arrakis, applications send and receive network packets by directly communicating with hardware. The data plane interface is therefore implemented in an application library, allowing it to be codesigned with the application [Marinos et al. 2014]. The Arrakis library provides two interfaces to applications. We describe the native Arrakis interface, which departs slightly from the POSIX standard to support true zero-copy I/O; Arrakis also provides a POSIX compatibility layer that supports unmodified applications.

Applications send and receive packets on queues, which have previously been assigned filters as described earlier. While filters can include IP, TCP, and UDP field predicates, Arrakis does not require the hardware to perform protocol processing, only multiplexing. In our implementation, Arrakis provides a user-space network stack above the data plane interface. This stack is designed to maximize both latency and throughput. We maintain a clean separation between three aspects of packet transmission and reception.

First, packets are transferred asynchronously between the network and main memory using conventional DMA techniques using rings of packet buffer descriptors.

Second, the application transfers ownership of a transmit packet to the network hardware by enqueuing a chain of buffers onto the hardware descriptor rings, and acquires a received packet by the reverse process. This is performed by two VNIC driver functions. **send_packet(queue, packet_array)** sends a packet on a queue; the packet is specified by the scatter-gather array **packet_array** and must conform to a filter already associated with the queue. **receive_packet(queue) = packet** receives a packet from a queue and returns a pointer to it. Both operations are asynchronous. **packet_done(packet)** returns ownership of a received packet to the VNIC.

For optimal performance, the Arrakis stack would interact with the hardware queues not through these calls but directly via compiler-generated, optimized code tailored to the NIC descriptor format. However, the implementation we report on in this article uses function calls to the driver.

Third, we handle asynchronous notification of events using doorbells associated with queues. Doorbells are delivered directly from hardware to user programs via hardware virtualized interrupts when applications are running and via the control plane to invoke the scheduler when applications are not running. In the latter case, higher latency is tolerable. Doorbells are exposed to Arrakis programs via regular event delivery mechanisms (e.g., a file descriptor event) and are fully integrated with existing I/O multiplexing interfaces (e.g., `select`). They are useful both to notify an application of general availability of packets in receive queues and as a lightweight notification mechanism for I/O completion and the reception of packets in high-priority queues.

This design results in a protocol stack that decouples hardware from software as much as possible using the descriptor rings as a buffer, maximizing throughput and minimizing overhead under high packet rates, yielding low latency. On top of this native interface, Arrakis provides POSIX-compatible sockets. This compatibility layer allows Arrakis to support unmodified Linux applications. However, we show that performance gains can be achieved by using the asynchronous native interface.

3.7. Storage Data Plane Interface

The low-level storage API provides a set of commands to asynchronously read, write, and flush hardware caches at any offset and of arbitrary size in a VSA via a command queue in the associated VSIC. To do so, the caller provides an array of virtual memory ranges (address and size) in RAM to be read/written, the VSA identifier, the queue number, and the matching array of ranges (offset and size) within the VSA. The implementation enqueues the corresponding commands to the VSIC, coalescing and reordering commands if this makes sense to the underlying media. I/O completion events are reported using doorbells. On top of this, a POSIX-compliant file system is provided.

We have also designed a library of *persistent data structures*, Caladan, to take advantage of low-latency storage devices. Persistent data structures can be more efficient than a simple read/write interface provided by file systems. Their drawback is a lack of backward compatibility to the POSIX API. Our design goals for persistent data structures are that (1) operations are immediately persistent, (2) the structure is robust versus crash failures, and (3) operations have minimal latency.

We have designed persistent log and queue data structures according to these goals and modified a number of applications to use them (e.g., Section 4.4). These data structures manage all metadata required for persistence, which allows tailoring of that data to reduce latency. For example, metadata can be allocated along with each data structure entry and persisted in a single hardware write operation. For the log and queue, the only metadata that needs to be kept is where they start and end. Pointers link entries to accommodate wraparounds and holes, optimizing for linear access and efficient prefetch of entries. By contrast, a file system typically has separate inodes to manage block allocation. The in-memory layout of Caladan structures is as stored, eliminating marshaling.

The log API includes operations to open and close a log, create log entries (for metadata allocation), append them to the log (for persistence), iterate through the log (for reading), and trim the log. The queue API adds a pop operation to combine trimming and reading the queue. Persistence is asynchronous: an append operation returns immediately with a callback on persistence. This allows us to mask remaining write latencies, for example, by optimistically preparing network responses to clients, while an entry is persisted.

Entries are allocated in multiples of the storage hardware's minimum transfer unit (MTU—512 bytes for our RAID controller, based on SCSI) and contain a header that denotes the true (byte-granularity) size of the entry and points to the offset of the next entry in a VSA. This allows entries to be written directly from memory, without additional marshaling. At the end of each entry is a marker that is used to determine whether an entry was fully written (empty VSA space is always zero). By issuing appropriate cache flush commands to the storage hardware, Caladan ensures that markers are written after the rest of the entry (cf. Chidambaram et al. [2013]).

Both data structures are identified by a header at the beginning of the VSA that contains a version number, the number of entries, the MTU of the storage device, and a pointer to the beginning and end of the structure within the VSA. Caladan repairs a corrupted or outdated header lazily in the background upon opening, by looking for additional, complete entries from the purported end of the structure.

3.8. Implementation

The Arrakis operating system is based on the Barrelfish [Baumann et al. 2009] multicore OS code base.[6] We added 33,786 lines of code to the Barrelfish code base in order to

[6]http://www.barrelfish.org/.

implement Arrakis. Our changes have been merged back and are available in the main Barrelfish source tree. Barrelfish lends itself well to our approach, as it already provides a library OS. We could have also chosen to base Arrakis on the Xen [Barham et al. 2003] hypervisor or the Intel Data Plane Development Kit (DPDK) [Intel Corporation 2013a] running on Linux; both provide user-level access to the network interface via hardware virtualization. However, implementing a library OS from scratch on top of a monolithic OS would have been more time consuming than extending the Barrelfish library OS.

We extended Barrelfish with support for SR-IOV, which required modifying the existing PCI device manager to recognize and handle SR-IOV extended PCI capabilities. We implemented a physical function driver for the Intel 82599 10G Ethernet Adapter [Intel Corporation 2010] that can initialize and manage a number of virtual functions. We also implemented a virtual function driver for the 82599, including support for Extended Message Signaled Interrupts (MSI-X), which are used to deliver per-VNIC doorbell events to applications. Finally, we implemented drivers for the Intel IOMMU [Intel Corporation 2013c] and the Intel RS3 family of RAID controllers [Intel Corporation 2013b]. In addition—to support our benchmark applications—we added several POSIX APIs that were not implemented in the Barrelfish code base, such as POSIX threads, many functions of the POSIX sockets' API, and the epoll interface found in Linux to allow scalable polling of a large number of file descriptors. For convenience in the short term, we place the control plane implementation in the 82599 physical function driver and export its API to applications via the Barrelfish interprocess communication system. As support for more devices is added to Arrakis, we will factor the generic control plane implementation out of the 82599 driver. Barrelfish already supports standalone user-mode device drivers, akin to those found in microkernels. We created shared library versions of the drivers, which we link to each application.

We have developed our own user-level network stack, Extaris . Extaris is a shared library that interfaces directly with the virtual function device driver and provides the POSIX sockets' API and Arrakis's native API to the application. Extaris is based in part on the low-level packet processing code of the lwIP network stack.[7] It has identical capabilities to lwIP but supports hardware offload of layer 3 and 4 checksum operations and does not require any synchronization points or serialization of packet operations. We have also developed our own storage API layer, as described in Section 3.7 and our library of persistent data structures, Caladan.

3.9. Limitations and Future Work

Due to the limited filtering support of the 82599 NIC, our implementation uses a different MAC address for each VNIC, which we use to direct flows to applications and then do more fine-grained filtering in software, within applications. The availability of more general-purpose filters would eliminate this software overhead.

Our implementation of the virtual function driver does not currently support the "transmit descriptor head writeback" feature of the 82599, which reduces the number of PCI bus transactions necessary for transmit operations. We expect to see a 5% network performance improvement from adding this support.

The RS3 RAID controller we used in our experiments does not support SR-IOV or VSAs. Hence, we use its physical function, which provides one hardware queue, and we map a VSA to each logical disk provided by the controller. We still use the IOMMU for protected access to application virtual memory, but the controller does not protect access to logical disks based on capabilities. Our experience with the 82599 suggests

[7]http://savannah.nongnu.org/projects/lwip/.

that hardware I/O virtualization incurs negligible performance overhead versus direct access to the physical function. We expect this to be similar for storage controllers.

Fine-Grained I/O Scheduling. Unmediated device access requires some I/O scheduling to be carried out in hardware. Current hardware technology offers relatively coarse-grained scheduling, such as rate limiting and queue priorities at the virtual controller I/O level. Fine-grained scheduling, such as network congestion control, is left with individual application-level I/O stacks. However, within a multitenant data center, it is not appropriate to trust applications, or even virtual machines, with their own congestion control. This poses the question of how to control I/O scheduling of untrusted I/O stacks. There are various ways to address this question.

The first way, taken by the IX [Belay et al. 2014] operating system, is to again protect the application-level I/O stack so that it cannot be modified by the application developer. This comes at a performance cost due to the protection boundary and also restricts application developers to use a set of I/O stacks that have been designated “safe for use” by the cloud provider.

A second way is to add more fine-grained I/O control to hardware I/O devices. For example, we have proposed a more flexible network device that is able to perform fine-grained I/O scheduling based on a packet match and action paradigm [Kaufmann et al. 2015]. The NIC can perform limited interactions with transport-level network protocols and use this capability to enforce congestion control and other fine-grained I/O scheduling tasks at the hardware level. This approach does not impose performance overheads or rely on designated network I/O stacks. On the other hand, network protocols need to fit the match and action paradigm to allow fine-grained hardware I/O scheduling.

A third way is to integrate Arrakis with a distributed data center network resource allocator. Earlier work has shown that distributed rate limiters can provide the basis for an efficient and fair congestion control mechanism for shared bottlenecks [Raghavan et al. 2007]. Although scalability of the mechanism might be a concern, almost all congestion in data centers occurs for access to the top of rack switch [Halperin et al. 2011]. As a result, a distributed controller might be feasible. This approach promises to integrate well with flexible hardware I/O controllers to provide high-performance congestion control of untrusted applications for relevant scenarios in multitenant data centers.

Congestion can, of course, also occur in the broader Internet, for example, at the client access link. Typically in this case, the application is as interested as anyone in not overrunning the bottleneck. For example, Chrome has introduced a delay-based user-level congestion control protocol for its UDP traffic [Roskind 2013]. If abuse becomes common, the Arrakis control plane can use rate limiters to enforce bounds on application behavior, for example, by probing the path to the client on connection setup.

4. EVALUATION

We evaluate Arrakis on four cloud application workloads: a typical, read-heavy load pattern observed in many large deployments of the memcached distributed object caching system; a write-heavy load pattern to the Redis persistent NoSQL store; a workload consisting of a large number of individual client HTTP requests made to a farm of web servers via an HTTP load balancer; and finally, the same benchmark via an IP-layer middlebox. We also examine the system under maximum load in a series of microbenchmarks and analyze performance crosstalk among multiple networked applications. Using these experiments, we seek to answer the following questions:

- What are the major contributors to performance overhead in Arrakis and how do they compare to those of Linux (presented in Section 2)?

- Does Arrakis provide better latency and throughput for real-world cloud applications? How does the throughput scale with the number of CPU cores for these workloads?
- Can Arrakis retain the benefits of user-level application execution and kernel enforcement while providing high-performance packet-level network IO?
- What additional performance gains are possible by departing from the POSIX interface?

We compare the performance of the following OS configurations: Linux kernel version 3.8 (Ubuntu version 13.04), Arrakis using the POSIX interface (Arrakis/P), and Arrakis using its native interface (Arrakis/N).

We tuned Linux network performance by installing the latest ixgbe device driver version 3.17.3 and disabling receive-side scaling (RSS) when applications execute on only one processor. RSS spreads packets over several NIC receive queues but incurs needless coherence overhead on a single core. The changes yield a throughput improvement of 10% over nontuned Linux. We use the kernel-shipped MegaRAID driver version 6.600.18.00-rc1.

Linux uses a number of performance-enhancing features of the network hardware, which Arrakis does not currently support. Among these features is the use of direct processor cache access by the NIC, TCP and UDP segmentation offload, large receive offload, and network packet header splitting. All of these features can be implemented in Arrakis ; thus, our performance comparison is weighted in favor of Linux.

4.1. Server-Side Packet Processing Performance

We load the UDP echo benchmark from Section 2 on the server and use all other machines in the cluster as load generators. These generate 1KB UDP packets at a fixed rate and record the rate at which their echoes arrive. Each experiment exposes the server to maximum load for 20 seconds.

Shown in Table I, compared to Linux, Arrakis eliminates two system calls, software demultiplexing overhead, socket buffer locks, and security checks. In Arrakis/N, we additionally eliminate two socket buffer copies. Arrakis/P incurs a total server-side overhead of 1.44μs, 57% less than Linux. Arrakis/N reduces this overhead to 0.38μs.

The echo server is able to add a configurable delay before sending back each packet. We use this delay to simulate additional application-level processing time at the server. Figure 7 shows the average throughput attained by each system over various such delays; the theoretical line rate is 1.26M pps with zero processing.

In the best case (no additional processing time), Arrakis/P achieves 2.3× the throughput of Linux. By departing from POSIX, Arrakis/N achieves 3.9× the throughput of Linux. The relative benefit of Arrakis disappears at 64μs. To gauge how close Arrakis comes to the maximum possible throughput, we embedded a minimal echo server directly into the NIC device driver, eliminating any remaining API overhead. Arrakis/N achieves 94% of the driver limit.

4.2. Memcached Key-Value Store

Memcached is an in-memory key-value store used by many cloud applications. It incurs a processing overhead of 2 to 3μs for an average object fetch request, comparable to the overhead of OS kernel network processing.

We are not concerned with investigating potential end-to-end latency improvements. We only require the memcached server to run the Arrakis operating system and assume that the client machines remain unmodified. In this scenario, we do not expect end-to-end transaction latency to improve by a significant margin, as the overheads of the network and client networking stacks remain unmodified.

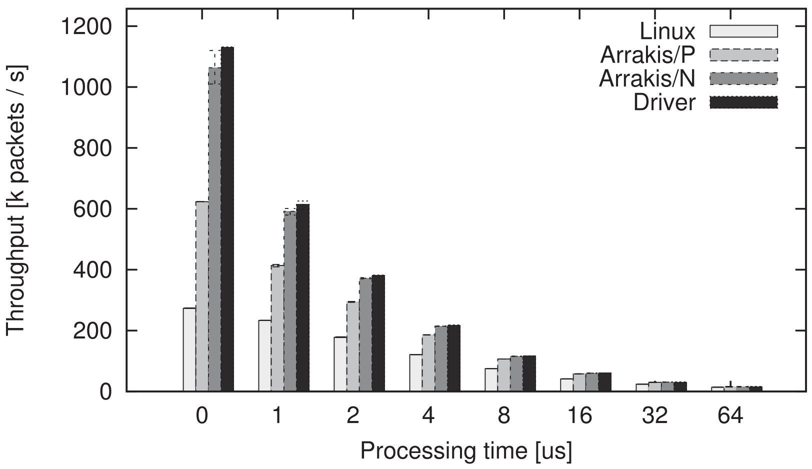

Fig. 7. Average UDP echo throughput for packets with 1,024-byte payload over various processing times. The top y-axis value shows theoretical maximum throughput on the 10G network. Error bars in this and following figures show min/max measured over five repeats of the experiment.

We benchmark memcached 1.4.15 by sending it requests at a constant rate via its binary UDP protocol, using a tool similar to the popular memslap benchmark.[8] We configure a workload pattern of 90% fetch and 10% store requests on a pregenerated range of 128 different keys of a fixed size of 64 bytes and a value size of 1KB, in line with real cloud deployments [Atikoglu et al. 2012].

To measure network stack scalability for multiple cores, we vary the number of memcached server processes. Each server process executes independently on its own port number, such that measurements are not impacted by scalability bottlenecks in memcached itself, and we distribute load equally among the available memcached instances. On Linux, memcached processes share the kernel-level network stack. On Arrakis, each process obtains its own VNIC with an independent set of packet queues, each controlled by an independent instance of Extaris .

Figure 8 shows that memcached on Arrakis/P achieves 1.7× the throughput of Linux on one core and attains near line-rate at four CPU cores. The slightly lower throughput on all six cores is due to contention with Barrelfish system management processes, such as memory and process managers, that consistently poll intercore messaging channels and are pinned to that core, causing high CPU utilization [Baumann et al. 2009]. By contrast, Linux throughput nearly plateaus beyond two cores. A single, multithreaded memcached instance shows no noticeable throughput difference to the multiprocess scenario. This is not surprising as memcached is optimized to scale well.

To conclude, the separation of network stack and application in Linux provides only limited information about the application's packet processing and poses difficulty assigning threads to the right CPU core. The resulting cache misses and socket lock contention are responsible for much of the Linux overhead. In Arrakis, the application is in control of the whole packet processing flow: assignment of packets to packet queues and packet queues to cores, and finally the scheduling of its own threads on these cores. The network stack thus does not need to acquire any locks, and packet data is always available in the right processor cache.

Memcached is also an excellent example of the communication endpoint abstraction: we can create hardware filters to allow packet reception and transmission only between

[8]http://www.libmemcached.org/.

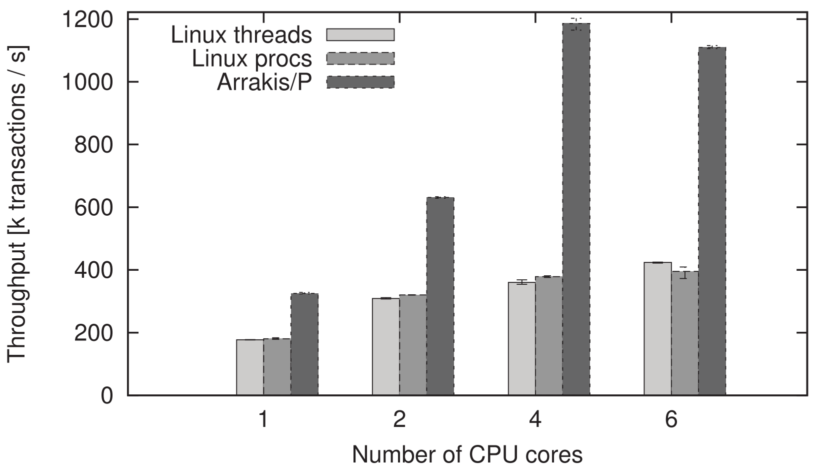

Fig. 8. Average memcached transaction throughput and scalability. Top y-axis value = 10Gb/s.

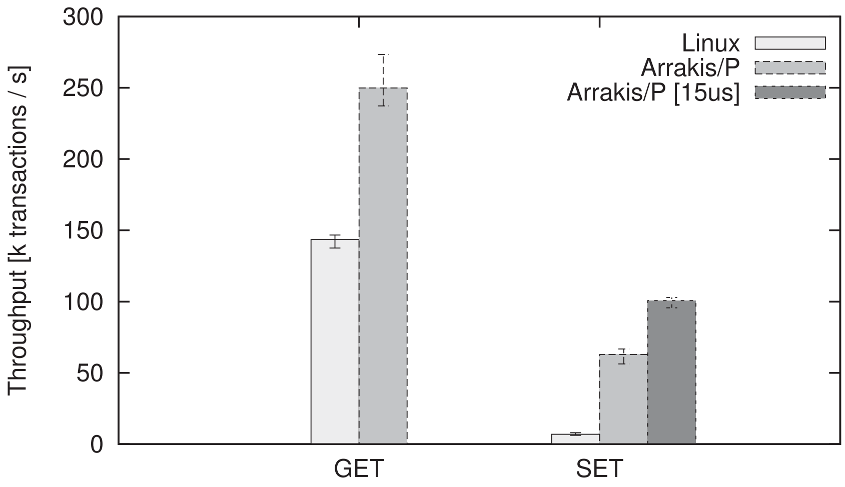

Fig. 9. Average Redis transaction throughput for GET and SET operations. The Arrakis/P [15us] and Linux/Caladan configurations apply only to SET operations.

the memcached server and a designated list of client machines that are part of the cloud application. In the Linux case, we have to filter connections in the application.

4.3. Arrakis Native Interface Case Study

As a case study, we modified memcached to make use of Arrakis/N. In total, 74 lines of code were changed, with 11 pertaining to the receive side and 63 to the send side. On the receive side, the changes involve eliminating memcached's receive buffer and working directly with pointers to packet buffers provided by Extaris, as well as returning completed buffers to Extaris . The changes increase average throughput by 9% over Arrakis/P. On the send side, changes include allocating a number of send buffers to allow buffering of responses until fully sent by the NIC, which now must be done within memcached itself. They also involve the addition of reference counts to hash table entries and send buffers to determine when it is safe to reuse buffers and hash table entries that might otherwise still be processed by the NIC. We gain an additional 10% average throughput when using the send-side API in addition to the receive-side API.

We conclude that decent performance improvements can be attained with modest application changes, by moving to an API that eliminates packet copy overheads. We note that this API change is only useful with unmediated application access to the network device. A lightweight asynchronous notification mechanism is required, as well as the ability to configure network queues to steer packet data to the right CPU cache and align it properly according to the application's needs.

4.4. Redis NoSQL Store

Redis extends the memcached model from a cache to a persistent NoSQL object store. Our results in Table II show that Redis operations—while more laborious than memcached—are still dominated by I/O stack overheads.

Redis can be used in the same scenario as memcached and we follow an identical experiment setup, using Redis version 2.8.5. We use the benchmarking tool distributed with Redis and configure it to execute GET and SET requests in two separate benchmarks to a range of 65,536 random keys with a value size of 1,024 bytes, persisting each SET operation individually, with a total concurrency of 1,600 connections from 16 benchmark clients executing on the client machines. Redis is single threaded, so we investigate only single-core performance.

The Arrakis version of Redis uses Caladan. We changed 109 lines in the application to manage and exchange records with the Caladan log instead of a file. We did not eliminate Redis's marshaling overhead (cf. Table II). If we did, we would save another 2.43μs of write latency. Due to the fast I/O stacks, Redis's read performance mirrors that of memcached and write latency improves by 63%, while write throughput improves vastly, by 9$\times$.

To investigate what would happen if we had access to state-of-the-art storage hardware, we simulate (via a write-delaying RAM disk) a storage back end with 15μs write latency, such as the ioDrive2 [Fusion-IO 2014]. Write throughput improves by another 1.6$\times$, nearing Linux read throughput.

Both network and disk virtualization are needed for good Redis performance. We tested this by porting Caladan to run on Linux, with the unmodified Linux network stack. This improved write throughput by only 5$\times$ compared to Linux, compared to 9$\times$ on Arrakis.

Together, the combination of data-plane network and storage stacks can yield large benefits in latency and throughput for both read- and write-heavy workloads. The tight integration of storage and data structure in Caladan allows for a number of latency-saving techniques that eliminate marshaling overhead and bookkeeping of journals for file system metadata, and can offset storage allocation overhead. These benefits will increase further with upcoming hardware improvements.

4.5. HTTP Load Balancer

To aid scalability of web services, HTTP load balancers are often deployed to distribute client load over a number of web servers. A popular HTTP load balancer employed by many web and cloud services, such as Amazon EC2 and Twitter, is haproxy.[9] In these settings, many connections are constantly opened and closed and the OS needs to handle the creation and deletion of the associated socket data structures.

[9]http://haproxy.1wt.eu.

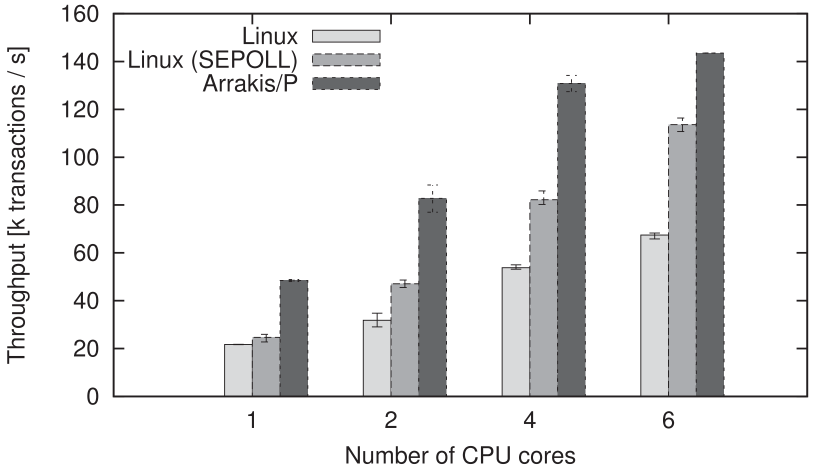

Fig. 10. Average HTTP transaction throughput and scalability of haproxy.

To investigate how performance is impacted when many connections need to be maintained, we configure five web servers and one load balancer. To minimize overhead at the web servers, we deploy a simple static web page of 1,024 bytes, served out of main memory. These same web server hosts also serve as workload generators, using ApacheBench version 2.3 to conduct as many concurrent requests for the web page as possible. Each request is encapsulated in its own TCP connection. On the load balancer host, we deploy haproxy version 1.4.24, configured to distribute incoming load in a round-robin fashion. We run multiple copies of the haproxy process on the load-balancing node, each executing on its own port number. We configure the ApacheBench instances to distribute their load equally among the available haproxy instances.

Haproxy relies on cookies, which it inserts into the HTTP stream to remember connection assignments to back-end web servers under possible client reconnects. This requires it to interpret the HTTP stream for each client request. Linux provides an optimization called TCP splicing that allows applications to forward traffic between two sockets without user-space involvement. This reduces the overhead of kernel crossings when connections are long-lived. We enable haproxy to use this feature on Linux when beneficial.

Finally, haproxy contains a feature known as “speculative epoll” (SEPOLL), which uses knowledge about typical socket operation flows within the Linux kernel to avoid calls to the epoll interface and optimize performance. Since the Extaris implementation differs from that of the Linux kernel network stack, we were not able to use this interface on Arrakis, but speculate that this feature could be ported to Arrakis to yield similar performance benefits. To show the effect of the SEPOLL feature, we repeat the Linux benchmark both with and without it and show both results.

In Figure 10, we can see that Arrakis outperforms Linux in both regular and SEPOLL configurations on a single core, by a factor of 2.2 and 2, respectively. Both systems show similar scalability curves. Note that Arrakis’s performance on six CPUs is affected by background activity on Barrelfish.

To conclude, connection-oriented workloads require a higher number of system calls for setup (`accept` and `setsockopt`) and teardown (`close`). In Arrakis, we can use filters, which require only one control plane interaction to specify which clients and servers may communicate with the load balancer service. Further socket operations are reduced to function calls in the library OS, with lower overhead.

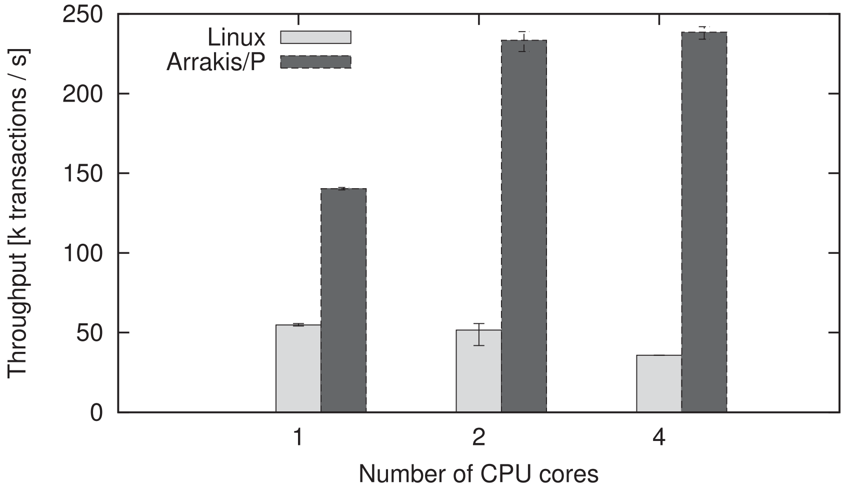

Fig. 11. Average HTTP transaction throughput and scalability of the load-balancing middlebox. Top y-axis value = 10Gb/s.

4.6. IP-Layer Middlebox

IP -layer middleboxes are ubiquitous in today's wide area networks (WANs). Common middleboxes perform tasks, such as firewalling, intrusion detection, network address translation, and load balancing. Due to the complexity of their tasks, middleboxes can benefit from the programming and runtime convenience provided by an OS through its abstractions for safety and resource management.

We implemented a simple user-level load balancing middlebox using raw IP sockets.[10] Just like haproxy, the middlebox balances an incoming TCP workload to a set of back-end servers. Unlike haproxy, it is operating completely transparently to the higher-layer protocols. It simply rewrites source and destination IP addresses and TCP port numbers contained in the packet headers. It monitors active TCP connections and uses a hash table to remember existing connection assignments. Responses by the back-end web servers are also intercepted and forwarded back to the corresponding clients. This is sufficient to provide the same load-balancing capabilities as in the haproxy experiment. We repeat the experiment from Section 4.5, replacing haproxy with our middlebox.

The simpler nature of the middlebox is reflected in the throughput results (see Figure 11). Both Linux and Arrakis perform better. Because the middlebox performs less application-level work than haproxy, performance factors are largely due to OS-level network packet processing. As a consequence, Arrakis's benefits are more prominent, and its performance is 2.6× that of Linux. We also see an interesting effect: the Linux implementation does not scale at all in this configuration. The reason for this is the raw IP sockets, which carry no connection information. Without an indication of which connections to steer to which sockets, each middlebox instance has to look at each incoming packet to determine whether it should handle it. This added overhead outweighs any performance gained via parallelism. In Arrakis, we can configure the hardware filters to steer packets based on packet header information and thus scale until we quickly hit the NIC throughput limit at two cores.

We conclude that Arrakis allows us to retain the safety, abstraction, and management benefits of software development at user level while vastly improving the performance

[10]http://man7.org/linux/man-pages/man7/raw.7.html.

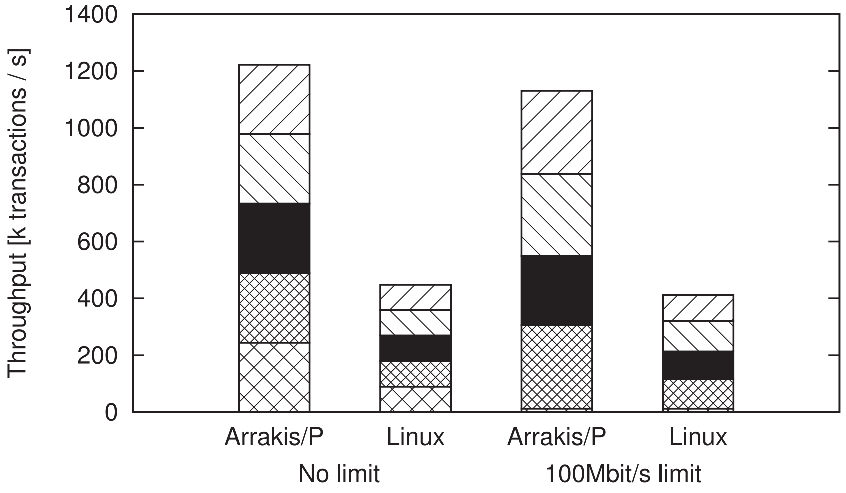

Fig. 12. Memcached transaction throughput over five instances (patterns), with/without rate limiting.

of low-level packet operations. Filters provide a versatile interface to steer packet workloads based on arbitrary information stored in packet headers to effectively leverage multicore parallelism, regardless of protocol specifics.

4.7. Performance Isolation

We show that QoS limits can be enforced in Arrakis, by simulating a simple multitenant scenario with five memcached instances pinned to distinct cores, to minimize processor crosstalk. One tenant has an SLA that allows it to send up to 100Mb/s. The other tenants are not limited.

We use rate specifiers in Arrakis to set the transmit rate limit of the VNIC of the limited process. On Linux, we use queuing disciplines [Hubert 2009] (specifically, HTB [Devera 2002]) to rate-limit the source port of the equivalent process.

We repeat the experiment from Section 4.2, plotting the throughput achieved by each memcached instance, shown in Figure 12. The bottom-most process (barely visible) is rate-limited to 100Mb/s in the experiment shown on the right-hand side of the figure. All runs remained within the error bars shown in Figure 8. When rate-limiting, a bit of the total throughput is lost for both OSs because clients keep sending packets at the same high rate. These consume network bandwidth, even when later dropped due to the rate limit.

We conclude that it is possible to provide the same kind of QoS enforcement—in this case, rate-limiting—in Arrakis, as in Linux. Thus, we are able to retain the protection and policing benefits of user-level application execution while providing improved network performance.

5. DISCUSSION

In this section, we discuss how we can extend the Arrakis model to apply to virtualized guest environments, as well as to interprocessor interrupts.

5.1. Arrakis as Virtualized Guest

Arrakis's model can be extended to virtualized environments. Making Arrakis a host in this environment is straightforward—this is what the technology was originally designed for. The best way to support Arrakis as a guest is by moving the control plane into the virtual machine monitor (VMM). Arrakis guest applications can then allocate virtual interface cards directly from the VMM. A simple way of accomplishing this is by preallocating a number of virtual interface cards in the VMM to the guest and letting applications pick only from this preallocated set, without requiring a special interface to the VMM.

The hardware limits apply to a virtualized environment in the same way as they do in the regular Arrakis environment. We believe the current limits on virtual adapters (typically 64) to be balanced with the number of available processing resources.

5.2. Virtualized Interprocessor Interrupts

To date, most parallel applications are designed assuming that shared memory is (relatively) efficient, while interprocessor signaling is (relatively) inefficient. A cache miss to data written by another core is handled in hardware, while alerting a thread on another processor requires kernel mediation on both the sending and receiving side. The kernel is involved even when signaling an event between two threads running inside the same application.

With kernel bypass, a remote cache miss and a remote event delivery are similar in cost at a physical level. Modern hardware already provides the operating system the ability to control how device interrupts are routed. To safely deliver an interrupt within an application, without kernel mediation, requires that the hardware add access control. With this, the kernel could configure the interrupt routing hardware to permit signaling among cores running the same application, trapping to the kernel only when signaling between different applications.

6. RELATED WORK

SPIN [Bershad et al. 1995] and Exokernel [Ganger et al. 2002] reduced shared kernel components to allow each application to have customized operating system management. Nemesis [Black et al. 1997] reduced shared components to provide more performance isolation for multimedia applications. All three mediated I/O in the kernel. Relative to these systems, Arrakis shows that application customization is consistent with very high performance.

Following U-Net, a sequence of hardware standards such as VIA [Compaq Computer Corp. et al. 1997] and Infiniband [Infiniband Trade Organization 2010] addressed the challenge of minimizing, or eliminating entirely, operating system involvement in sending and receiving network packets in the common case. To a large extent, these systems have focused on the needs of parallel applications for high-throughout, low-overhead communication. Arrakis supports a more general networking model including client-server and peer-to-peer communication.

Our work was inspired in part by previous work on Dune [Belay et al. 2012], which used nested paging to provide support for user-level control over virtual memory, and Exitless IPIs [Gordon et al. 2012], which presented a technique to demultiplex hardware interrupts between virtual machines without mediation from the virtual machine monitor.

Netmap [Rizzo 2012] implements high-throughput-network I/O by doing DMAs directly from user space. Sends and receives still require system calls, as the OS needs to do permission checks on every operation. Throughput is achieved at the expense of latency, by batching reads and writes. Similarly, IX [Belay et al. 2014] implements a custom, per-application network stack in a protected domain accessed with batched system calls. Arrakis eliminates the need for batching by handling operations at the user level in the common case.

Concurrently with our work, mTCP uses Intel's DPDK interface to implement a scalable user-level TCP [Jeong et al. 2014]; mTCP focuses on scalable network stack design, while our focus is on the operating system API for general client-server applications.

We expect the performance of Extaris and mTCP to be similar. OpenOnload[11] is a hybrid user- and kernel-level network stack. It is completely binary compatible with existing Linux applications; to support this, it has to keep a significant amount of socket state in the kernel and supports only a traditional socket API. Arrakis, in contrast, allows applications to access the network hardware directly and does not impose API constraints.

Recent work has focused on reducing the overheads imposed by traditional file systems and block device drivers, given the availability of low-latency persistent memory. DFS [Josephson et al. 2010] and PMFS [Dulloor et al. 2014] are file systems designed for these devices. DFS relies on the flash storage layer for functionality traditionally implemented in the OS, such as block allocation. PMFS exploits the byte addressability of persistent memory, avoiding the block layer. Both DFS and PMFS are implemented as kernel-level file systems, exposing POSIX interfaces. They focus on optimizing file system and device driver design for specific technologies, while Arrakis investigates how to allow applications fast, customized device access.

Moneta-D [Caulfield et al. 2012] is a hardware and software platform for fast, user-level I/O to solid-state devices. The hardware and operating system cooperate to track permissions on hardware extents, while a user-space driver communicates with the device through a virtual interface. Applications interact with the system through a traditional file system. Moneta-D is optimized for large files, since each open operation requires communication with the OS to check permissions; Arrakis does not have this issue, since applications have complete control over their VSAs. Aerie [Volos et al. 2014] proposes an architecture in which multiple processes communicate with a trusted user-space file system service for file metadata and lock operations while directly accessing the hardware for reads and data-only writes. Arrakis provides more flexibility than Aerie, since storage solutions can be integrated tightly with applications rather than provided in a shared service, allowing for the development of higher-level abstractions, such as persistent data structures.

7. CONCLUSION

In this article, we described and evaluated Arrakis, a new operating system designed to remove the kernel from the I/O data path without compromising process isolation. Unlike a traditional operating system, which mediates all I/O operations to enforce process isolation and resource limits, Arrakis uses device hardware to deliver I/O directly to a customized user-level library. The Arrakis kernel operates in the control plane, configuring the hardware to limit application misbehavior.

To demonstrate the practicality of our approach, we have implemented Arrakis on commercially available network and storage hardware and used it to benchmark several typical server workloads. We are able to show that protection and high performance are not contradictory: end-to-end client read and write latency to the Redis persistent NoSQL store is 2–5× faster and write throughput 9× higher on Arrakis than on a well-tuned Linux implementation.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Emmett Witchel, for their comments and feedback. We also thank Oleg Godunok for implementing the IOMMU driver, Antoine Kaufmann for implementing MSI-X support, and Taesoo Kim for implementing interrupt support into Extaris.

[11]http://www.openonload.org/.

REFERENCES

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

Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS 2012*.

Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*.

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*.

Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*.

A. Bensoussan, C. T. Clingen, and R. C. Daley. 1972. The multics virtual memory: Concepts and design. *Communications of the ACM* 15 (1972), 308–318.

Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*.

Richard Black, Paul T. Barham, Austin Donnelly, and Neil Stratford. 1997. Protocol implementation in a vertically structured operating system. In *Proceedings of the 22nd Annual Conference on Local Computer Networks*.

Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*.

Compaq Computer Corp., Intel Corporation, and Microsoft Corporation. 1997. *Virtual Interface Architecture Specification* (version 1.0 ed.).

RDMA Consortium. 2009. Architectural Specifications for RDMA over TCP/IP. Retrieved from http://www.rdmaconsortium.org/.

Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Communications of the ACM* 56, 2 (Feb. 2013), 74–80.

Martin Devera. 2002. HTB Linux queuing discipline manual – User Guide. Retrieved from http://luxik.cdi.cz/devik/qos/htb/userg.pdf.

Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*.

Peter Druschel, Larry Peterson, and Bruce Davie. 1994. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications*.

Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th ACM SIGOPS/EuroSys European Conference on Computer Systems*.

Fusion-IO. 2014. *ioDrive2 and ioDrive2 Duo Multi Level Cell*. Fusion-IO. Product Datasheet. Retrieved from http://www.fusionio.com/load/-media-/2rezss/docsLibrary/FIO_DS_ioDrive2.pdf.

Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceño, Russell Hunt, and Thomas Pinckney. 2002. Fast and flexible application-level networking on Exokernel systems. *ACM Transactions on Computer Systems* 20, 1 (Feb. 2002), 49–83.

Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Whetherall. 2011. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM Conference*.

Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*.

Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2011. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*.

Bert Hubert. 2009. Linux Advanced Routing & Traffic Control HOWTO. Retrieved from http://www.lartc.org/howto/.

Infiniband Trade Organization. 2010. Introduction to Infiniband for End Users. Retrieved from https://cw.infinibandta.org/document/dl/7268.

Intel Corporation. 2010. *Intel 82599 10 GbE Controller Datasheet*. Revision 2.6. Retrieved from http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

Intel Corporation. 2013a. *Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide*. Intel Corporation. Reference Number: 326003-003.

Intel Corporation 2013b. *Intel RAID Controllers RS3DC080 and RS3DC040*. Intel Corporation. Product Brief. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/raid-controller-rs3dc-brief.pdf.

Intel Corporation. 2013c. *Intel Virtualization Technology for Directed I/O Architecture Specification*. Technical Report Order Number: D51397-006. Intel Corporation.

Intel Corporation. 2013d. *NVM Express* (revision 1.1a ed.). Intel Corporation. http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1a.pdf.

EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeongand Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*.

William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. 2010. DFS: A file system for virtualized flash storage. *Transactions on Storage* 6, 3, Article 14 (Sept. 2010), 14:1–14:25 pages.

Antoine Kaufmann, Simon Peter, Thomas E. Anderson, and Arvind Krishnamurthy. 2015. FlexNIC: Rethinking network DMA. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*.

P. Kutch. 2011. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. *Intel Application Note* 321211–002 (Jan. 2011).

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

Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the 5th Symposium on Cloud Computing*.

LSI Corporation 2010. *LSISAS2308 PCI Express to 8-Port 6Gb/s SAS/SATA Controller*. LSI Corporation. Product Brief. Retrieved from http://www.lsi.com/downloads/Public/SAS%20ICs/LSI_PB_SAS2308.pdf.

LSI Corporation 2014. *LSISAS3008 PCI Express to 8-Port 12Gb/s SAS/SATA Controller*. LSI Corporation. Product Brief. http://www.lsi.com/downloads/Public/SAS%20ICs/LSI_PB_SAS3008.pdf.

Ilias Marinos, Robert N. M. Watson, and Mark Handley. 2014. Network stack specialization for performance. In *Proceedings of the ACM SIGCOMM Conference*.

David Mosberger and Larry L. Peterson. 1996. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*.

Vivek S. Pai, Peter Druschel, and Willy Zwanepoel. 1999. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*.

Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving network connection locality on multicore systems. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*.

Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for end-host rate limiting. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*.

Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. 2007. Cloud control with distributed rate limiting. In *Proceedings of the ACM SIGCOMM Conference*.

Luigi Rizzo. 2012. Netmap: A novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference*.

Jim Roskind. 2013. Experimenting with QUIC. Retrieved from http://blog.chromium.org/2013/06/experimenting-with-quic.html.

Solarflare Communications, Inc. 2010. .Solarflare SFN5122F Dual-Port 10GbE Enterprise Server Adapter. Retrieved from http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_SFN5122F_10GbE_Adapter_Brief.pdf.

Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Roman Pletka, Blake G. Fitch, and Thomas R. Gross. 2013. Unified high-performance I/O: One stack to rule them all. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems*.

Haris Volos, Sanketh Nalli, Sankaralingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the 9th ACM SIGOPS/EuroSys European Conference on Computer Systems*.

T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*.

Received July 2015; accepted August 2015