Arrakis: The Operating System is the Control Plane

Simon Peter Jialin Li Irene Zhang
Dan R. K. Ports Arvind Krishnamurthy
Thomas Anderson
University of Washington

Timothy Roscoe
ETH Zurich

Abstract

Recent device hardware trends enable a new approach to the design of network servers. 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. The Arrakis kernel operates only in the control plane. We describe the the hardware and software changes needed to take advantage of this new abstraction, and we illustrate its power by showing significant latency and throughput improvements for network server applications relative to a well-tuned Linux implementation.

1. Introduction

This paper proposes to rearrange the division of labor between the operating system, application runtime library, and device hardware, in order to streamline performance for I/O-bound server applications. At the same time, we want to retain the security properties of a traditional operating system design.

The key features of an operating system—sandboxed execution, resource isolation, and virtualization of limited physical resources—all seem to require applications to operate at one level removed from hardware I/O devices. In a traditional operating system, the kernel mediates all access to I/O: every network packet, disk block, and interprocessor interrupt is handed from user-level to hardware, and vice versa, through the kernel. The kernel sanity checks arguments, enforces resource limits, and prevents buggy or malicious programs from evading the system's security policy.

Kernel mediation comes at a cost, however. The resulting performance is much less than what is possible from the raw hardware, particularly for I/O-intensive web services. Further, the kernel must provide a common implementation shared among all applications; being "all things to all people" makes its code paths longer and more general than the minimum necessary to support any individual application. The inevitable result: periodic calls that the operating system should "get out of the way" and give applications direct access to hardware devices. Nevertheless, most web services are still built as applications on top of a traditional kernel, because exchanging reliability for better performance is rarely a good tradeoff.

Our goal is to provide the best of both worlds. Taking a cue from very high speed Internet routers, we split the operating system into a separate control and data plane. On the data plane, with the right device hardware support, kernel mediation is not needed. Network packets, disk blocks, and processor interrupts can be safely routed directly to (and from) user-level without going through the kernel and without violating system security. The kernel *is* needed on the control plane: to configure which data paths are allowed and what resource limits are to be enforced in hardware.

An inspiration for our work is the recent development of sophisticated network device hardware for the virtual machine market [1, 19]. Without special hardware, the data path for a network packet on a virtual machine must traverse both the host kernel and the guest kernel before reaching the application server code. Today's state of the art device hardware elides one hop, providing a small number of "virtual" network devices that can be mapped directly into the guest operating system. The guest operating system can program these virtual network devices without host intermediation, by directly manipulating the virtual device transmit and receive queues.

We take this idea one step further: can we move the operating system off the I/O data path? We need hardware and software support for *safely* delivering I/O directly into a user-level library. In this paper, we focus on streamlining the data plane for network devices, as the available device hardware allows us to build a working prototype for that case. However, we note that our model is general-purpose enough to apply to other I/O devices, such as disks and interprocessor interrupts.

Our work is complementary but orthogonal to the nanokernel design pattern pioneered by several systems two decades ago. In this model, the operating system retains its sandboxing role and allocates resources to applications, but as much as possible, from that point forward the application is in complete control over how it uses its resources. User-level

virtual memory pagers in Mach [24], scheduler activations for multiprocessor management [2], and Exokernel disk management [18] all took this approach. The key idea in all of these systems is that the operating system remains free to change its allocation decisions, as long as it notifies the application. However, the nano-kernel movement did not envision changing the data path; the kernel was still involved in every I/O operation.

We make three specific contributions:

- We develop an architecture for the division of labor between the device hardware, operating system kernel, and runtime library for direct user-level I/O (Section 3).
- We implement a prototype of our model as a set of modifications to the open source Barrelfish operating system, running on commercially available multi-core computers and network device hardware (Section 3.5).
- We quantify the potential benefits of our approach for two widely used network services: a web object cache and an application-level load balancer (Section 4). We show that significant gains are possible in terms of both latency and scalability, relative to Linux, without modifying the application programming interface; additional gains are possible by tweaking the POSIX API.

2. Background

In this section, we give a detailed breakdown of the operating system overheads in networking stacks used today, followed by a discussion of new hardware technologies that enable Arrakis to almost completely eliminate operating systems overheads in the networking stack.

2.1 Inefficiencies in Networking Stacks

Operating system support for today's high speed network devices is imperfect. To motivate this work, we examine the sources of overhead in a traditional OS. 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.

To analyze the sources of overhead, we record timestamps at various stages of kernel and user-space processing of each packet. Our experiments are conducted using Ubuntu Linux 13.04 on a Intel Xeon E5-2430 (Sandy Bridge) system; further details of the hardware and benchmark configuration are presented in Section 4.

As shown in Table 1, operating system overhead for packet processing falls into four major categories.

- **Network stack** processing at the network card, IP, and UDP layers.
- **Scheduler overhead**: waking the appropriate process (if necessary), selecting it to run, and switching to its address space.

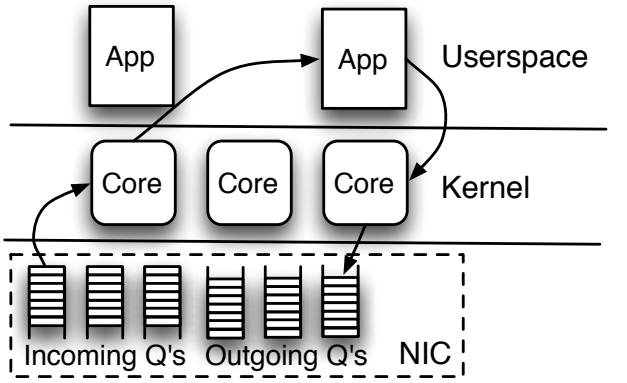

Figure 1. Linux networking architecture and workflow.

- **Kernel crossings**: from the 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 3.36 μs (see Table 1) spent processing each packet in Linux, nearly 70% is spent in the network stack. The work done in the network stack consists primarily of software demultiplexing and security checks. The kernel must validate the header of incoming packets, and must perform security checks on arguments provided by the application when it sends a packet.

Scheduler overhead depends significantly on whether the process in question 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 2.2 μs. In either case, making the transition from kernel to user mode and copying data to and from user-space buffers impose additional cost.

Another source of overhead is harder to directly quantify: cache and lock contention issues on multicore systems. These problems 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 (ARFS) aims to mitigate this cost, but these solutions themselves impose non-trivial setup costs [22].

By leveraging hardware support to remove kernel mediation from the data plane, Arrakis is able to eliminate certain categories of overhead entirely, and minimize the effect of others. Table 1 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 does not need to validate parameters provided by the user as extensively as a kernel implementation would. Because each application has a separate

		Linux				Arrakis			
		Process running		CPU idle		POSIX interface		Native interface	
Network stack	in	1.26	(37.6%)	1.24	(20.0%)	0.32	(22.1%)	0.21	(51.1%)
	out	1.05	(31.3%)	1.42	(22.9%)	0.28	(19.6%)	0.20	(48.9%)
Scheduler		0.17	(5.0%)	2.40	(38.8%)	-		-	
Copy	in	0.24	(7.1%)	0.25	(4.0%)	0.27	(18.4%)	-	
	out	0.44	(13.2%)	0.55	(8.9%)	0.58	(39.9%)	-	
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		0.41	

Table 1. Sources of packet processing overhead in Linux and Arrakis. All times are averages over 1000 samples, given in μs.

network stack, and packets are delivered to the cores where the application is running, lock contention and cache effects are reduced.

With Arrakis's optimized network stack, the time to copy packet data to and from user-provided buffers makes up the majority of the packet processing cost. This copying is required by the POSIX interface, which specifies that received packet data must be placed in a user-provided buffer, and permits buffers containing transmitted data to be reused by the application immediately. In addition to the POSIX compatibility interface, Arrakis provides a native interface which supports true zero-copy I/O.

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.

2.2 Hardware I/O Virtualization

Single-Root I/O Virtualization (SR-IOV) [19] is a hardware technology intended to support high-speed I/O for multiple virtual machines sharing a single physical machine. An SR-IOV-capable network adaptor 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 resembles a PCI network device, which can be directly mapped into a different virtual machine. System software with access to the physical function (such as Domain 0 in a Xen [4] installation) not only creates and deletes these virtual functions, but also configures filters in the SR-IOV adaptor to demultiplex incoming packets to different virtual functions.

In Arrakis, we use SR-IOV and virtualization hardware in a non-virtualized setting to completely remove any OS code from the data path. As we describe in the next section, existing SR-IOV adaptors [16, 26] do not provide all of the facilities needed for safe user-level operation; thus our implementation is a proof-of-concept, not a fully functional solution given the limitations of existing hardware.

3. Design and Implementation

The design of Arrakis is driven by the following design goals:

- **Minimize kernel involvement for data-plane operations:** Arrakis is designed to limit or completely eliminate kernel mediation for most I/O operations. Data packets are routed to and from the application's address space without requiring kernel involvement and without sacrificing security and isolation properties.
- **Transparency to application programmer:** Arrakis is designed to achieve significant performance improvements without requiring modifications to applications that are written assuming a POSIX API. Additional performance gains are possible if the developer can modify the application to take advantage of a native interface.
- **Right OS/hardware abstractions:** Arrakis's abstractions should provide sufficient flexibility to efficiently support a broad range of communication patterns, achieve the desired level of scalability on multi-core systems, and ably support application needs for locality and load balance.

In the rest of this section, we elaborate how we achieve these goals by presenting the Arrakis architecture. 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 programmer. Finally, we describe

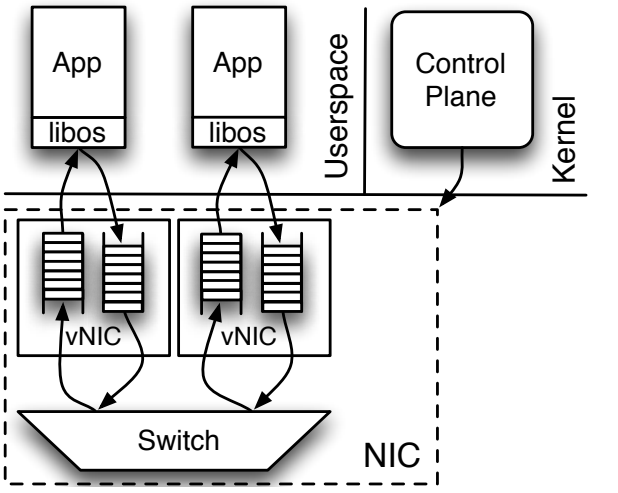

Figure 2. Arrakis networking architecture.

our implementation of Arrakis in the context of the Barrelfish operating system.

3.1 Architecture Overview

The overall architecture of Arrakis is depicted in Figure 2. Arrakis is aimed at I/O hardware that contain hardware support for virtualization. In this paper, we focus on networking 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 spaces, transmit/receive queues, and interrupts. The device exports a management interface that is accessible from the control plane in order to create or destroy virtualized device instances, associate individual instances with network flows, and allocate shared network resources to the different instances. Applications can transmit and receive network packets through a virtualized device instance without requiring kernel intervention. In order to perform these operations, applications rely on a user-level network stack, which is implemented as a library OS. The user-level network stack can be tailored to the application as it can assume exclusive access to a virtualized device instance and also avoid performing demultiplexing operations and security checks that are already implemented in the hardware.

3.2 Hardware model

In our design of Arrakis, we are aiming at 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 closely resembles what is already provided by many hardware virtualizable network adapters, and we hope it will itself provide guidance to future hardware designers.

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. Associated with each VNIC are *queues*, *filters*, and *rate limiters*. We discuss these components below.

Queues: Each VNIC contains multiple pairs of queues for user-space send and receive. The exact form of these VNIC queues could depend on the specifics of the network interface card. For example, it could support a scatter/gather interface to aggregate multiple physically-disjoint memory regions into a single data transfer. It could also optionally support hardware checksum offload and TCP segmentation facilities. These features enable packets to be handled more efficiently by performing additional work in hardware. In such cases, the Arrakis system offloads packet processing operations and thus reduces the software overhead associated with preparing these packets for transmission.

Transmit filters: a predicate on packets and packet header fields which the hardware will use to determine whether to send the packet or discard it (and 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 the need for kernel mediation to enforce these security checks.

Receive filters: a similar predicate which determines which packets received from the network will be delivered to a VNIC and to a specific queue associated with the target VNIC. The demultiplexing of packets to separate VNICs allows for isolation of flows and the further demultiplexing into queues allows for efficient load-balancing through techniques such as receive side scaling. Installation of transmit and receive filters are privileged operations performed via the control plane.

Bandwidth allocators: This includes support for resource allocation mechanisms such as rate limiters and pacing/traffic shaping of transmitted packets. Once a frame has been fetched out 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 inter-packet pacing controls associated with the queue. Installation of these controls are also privileged operations.

In addition, we assume that the NIC device driver supports an introspection interface that allows the control software to query for resource limits (e.g., the number of queues) and check for the availability of hardware support for packet processing (e.g., checksum calculations and segmentation).

Network cards that support SR-IOV incorporate the key elements of this model: they allow the creation of multiple VNICs that each may have multiple send and receive queues, and 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, 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.

3.3 Control plane interface

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

An application can create and delete VNICs (subject to resource limitations or policies imposed by Arrakis), and also associate *doorbells* with particular events on particular VNICs. A doorbell is an IPC communication end-point used to notify the application that an event (such as a packet arrival) has occurred, and we discuss them further in the next section.

Filters have a *type* (transmit or receive) and a *predicate* which corresponds to a convex sub-volume of the packet header space (for example, obtained with a set of mask-and-compare operations). Filters can be used to specify ranges of IP addresses and port numbers that can be 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 the communication patterns we want to be able to treat as a single unit, as well as subsuming traditional port allocation and interface specification.

To take one illustrative 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 end-point, but nevertheless isolate the data comprising the shuffle from other data. Furthermore, filters as subsets of header space are flexible enough to support a range of application settings while being more straightforward to implement in hardware than mechanisms such as longest prefix matching.

Applications create a filter with a control plane operation, and in practice this is usually wrapped in a higher-level call **create_filter(flags, peerlist, servicelist) = filter** which sacrifices generality in the aid of simplicity in the common case. It returns a new filter ID **filter**; **flags** specifies the filter direction (transmit or receive) and whether the filter refers to the Ethernet, IP, TCP, or UDP level. **peerlist** contains 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.

A filter ID is essentially a capability: it confers authority to send or receive packets satisfying its predicate. A filter ID can subsequently be **assign**ed to a particular queue on a VNIC; thereafter that queue can be used to send or receive the corresponding packets and the **assign** operation causes Arrakis to configure the underlying hardware accordingly.

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

3.4 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. The Arrakis library provides two interfaces to applications. We first 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 above. While filters can express packet properties which include IP, TCP, and UDP field predicates, Arrakis does not require the hardware to perform protocol processing, only multiplexing between filters. In the implementation we describe below, Arrakis provides a user-space network stack above the data place interface.

Interaction with the network hardware is designed for maximizing performance from the perspective of both latency and throughput. We borrow ideas from previous high-performance protocol stacks such as RDMA [12] and specialized HPC hardware, and maintain a clean separation between three aspects of packet transmission and reception.

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

Secondly, the application transfers ownership of a transmit packet to the network hardware by enqueuing a chain of buffer regions onto the same hardware descriptor rings, and acquires a received packet by the reverse process. At present, this is performed by two functions provided by the VNIC driver. **send_packet(queue, packet_array)** sends a packet on a queue. The packet is specified using the scatter-gather array **packet_array**, and must conform to some filter already associated with the queue. **receive_packet(queue) = packet** receives a packet from a queue and returns a pointer to it. Both these operations are asynchronous: they return immediately with a result code, regardless of whether the operation succeeds or not.

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 paper uses function calls to the driver.

Thirdly, we handle synchronous notification of events using doorbells associated with queues. These are exposed

to Arrakis programs as regular event delivery mechanisms (e.g., POSIX signals). Doorbells are useful both to notify an application of general availability of packets in receive queues, as well as a lightweight notification mechanism for the reception of packets in high-priority queues.

To facilitate waiting for multiple event types, a select-like interface to check for events on multiple queues at once is provided: **event_wait_queues(waitset)**, where waitset contains an array of **(queue, event, closure)** tuples. If an **event** on **queue** occurs, the corresponding **closure** is called with **queue** and **event** as arguments.

This design results in a protocol stack that decouples hardware from software as much as possible using the descriptor rings as a buffer, maximizing throughout and minimizing overhead under high packet rates, and achieving low latency by delivering doorbells directly from hardware to user programs via hardware virtualized interrupts.

On top of this native interface, Arrakis provides a POSIX-compatible sockets layer. This compatibility layer allows Arrakis to support unmodified Linux applications. However, performance gains can be achieved by using the native interface, which supports zero-copy I/O. The POSIX interface specifies that the application can reuse the buffers it provided to **sendmsg** calls immediately after the call returns, so the Arrakis POSIX compatibility layer must wait for these buffers to be flushed out to the network card. The native Arrakis interface instead provides an asynchronous notification to the application when the buffers may be safely overwritten and reused.

3.5 Implementation

The Arrakis operating system is based upon a fork of the Barrelfish [6] multicore OS code base.[1] We added 22,454 lines of code to the Barrelfish code base in order to implement Arrakis. Barrelfish lends itself well to our approach, as it already provides a library OS, which we can readily build upon. We could have chosen to base Arrakis on the Xen [4] hypervisor or even the Intel Data Plane Development Kit (DPDK) [17] running on Linux, both of which provide ways to gain direct 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 Arrakis with support for SR-IOV, which required modifying the existing PCI device manager to recognize and handle SR-IOV extended PCI capabilities, reserve PCI bus and memory address space to map PCI virtual functions, and implementing a physical function driver for the Intel 82599 10G Ethernet Adapter [16] that can initialize and manage a number of virtual functions. We also implemented a virtual function driver for the 82599 from scratch. 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, as well as the epoll interface found in Linux to allow scalable polling of a large number of file descriptors.

We place the control plane implementation in the 82599 physical function driver and export its API to applications via the Barrelfish inter-process communication system. The control plane implementation will likely be moved out of the 82599 driver and become more pervasive within Arrakis as support for more devices are added.

Barrelfish already supports standalone user-mode device drivers, akin to those found in microkernels. We created a shared library version of the virtual function driver, which we simply link to each application utilizing the network. The driver library automatically advertises its capability to handle a new virtual interface upon application startup and communicates with the physical function driver to invoke control plane operations, such as configuring the packet filters required to route packets to the appropriate VNICs.

We have developed our own user-level network stack, Arranet, which contains the data-plane implementation. Arranet is a shared library that interfaces directly with the virtual function device driver library and provides the POSIX sockets API and Arrakis's native API to the application. Arranet is based in part on the low-level packet processing code of the lightweight IP (lwIP) network stack [20]. 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. Instead, Arranet's high-level API calls operate directly on hardware-level packet queues via the device driver. For example, the **recvfrom** call will invoke the driver operation to directly poll one of the NIC's receive queues for the next packet, strip off the packet headers, and copy its payload into the user-provided receive buffer. Analogously, the **sendto** call will copy the provided data into pre-allocated send buffers and directly insert a corresponding packet descriptor into the NIC's send queue.

Prior to the development of Arranet, we experimented with lwIP, but quickly discovered that it is difficult to attain line-rate packet throughput at 10G Ethernet speeds or to handle tens of thousands of TCP connections when using its implementation of the POSIX socket API. The implementation of the POSIX socket API in lwIP is inherently non-scalable: every call is serialized and handled by a single, application-wide API handling thread.

We have integrated Arranet's implementation of packet flows with Barrelfish's messaging subsystem [5]. Arranet packet flows map directly to Barrelfish's channel abstraction. This makes it possible to re-use Barrelfish's messaging API to concurrently poll for a number of events on a variety of different channels.

[1] Publicly available at `http://www.barrelfish.org/`.

3.6 Limitations

The 82599 network device provides hardware support for standard 5-tuple filters that can be used to assign packet flows to receive queues within VNICs for scaling purposes. It however has only limited support for assigning flows to different VNICs. Only filters based on MAC addresses and VLAN tags are provided for this purpose. Hence, our implementation uses a different MAC address for each VNIC, which we use to direct flows to applications and then do more fine-grain filtering in software, within applications. The availability of more general-purpose filters would eliminate this software overhead.

Arrakis does not currently provide support for an IOMMU. In building our research prototype, we currently forego full hardware protection and write physical addresses directly to hardware registers. In a production environment, an IOMMU would be required. An IOMMU would also improve performance. Without it, Arrakis must manually translate virtual addresses for packet data to physical ones. Address translation makes up 5% of the total reported processing time in Table 1 for Arrakis/P and 15% of that reported for Arrakis/N. With proper IOMMU support, translations could be done faster in hardware.

Arranet does not currently support network interrupts, and all network packet processing is done by polling the network interface. This can have adverse effects on packet processing latency if network packets arrive when the network-handling application is not currently active. In such cases, Arrakis does not notice the arrival and does not allow the application to act immediately. Our benchmarks are not impacted by this limitation and support for network interrupts is future work.

Finally, our implementation of the virtual function driver does not currently support the transmission head writeback feature of 82599. If this was implemented, we would see a 5% network performance improvement.

4. Evaluation

We demonstrate the merits of Arrakis's approach via experimental evaluation of the performance of two cloud infrastructure applications: a web object cache and an HTTP load balancer. We also take a detailed look at the performance overheads associated with Arrakis by examining the system under maximum load in a series of microbenchmarks and compare the results to that of Linux. 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 2.1)?
- Does Arrakis achieve better latency and throughput for application-level packet processing than operating systems such as Linux that require kernel mediation for data plane operations?
- Does Arrakis provide better transaction throughput for real-world cloud application workloads? Also, how does the transaction throughput scale with the number of CPU cores for these workloads?
- What additional performance gains are possible by departing from the POSIX interface? Also, what changes are required to applications to take advantage of the native interface?

We conduct all of our experiments on a six machine cluster consisting of Intel Xeon E5-2430 (Sandy Bridge) systems. Each system's processor frequency is clocked to 2.2 GHz, with 1.5 MBytes total L2 cache and 15 MBytes total L3 cache, 4 GBytes of memory, and an Intel X520 dual-port 10 Gigabit Ethernet adapter (of which we use only one port). All machines are connected to a single 10 Gigabit Ethernet switch (a Dell PowerConnect 8024F). One system (the *server*) executes the application under scrutiny, while the others act as workload generators and consumers.

We compare the performance of the following OS configurations, which we deploy on the server:

- Ubuntu Server version 13.04 (Linux). This version of Ubuntu uses the Linux kernel version 3.8.
- Arrakis using the POSIX interface (Arrakis/P), and
- Arrakis using its native interface (Arrakis/N).

We tuned Linux's network performance by installing the latest ixgbe device driver version 3.17.3 and disabling receive side scaling when applications execute on only one processor. The changes yield a throughput improvement of 10% over non-tuned Linux.

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 and will likely yield additional performance improvements.

4.1 Server-side Packet Processing Latency

We test network stack performance with a simple UDP echo server that sequentially reads each incoming UDP packet in its entirety and then echoes it back to the sender, using non-blocking versions of the **recvfrom** and **sendto** POSIX socket calls. We use this benchmark to show the latency and bandwidth achieved by each system when all computation is focused on processing network packets.

We load the UDP echo benchmark on the server and use all other machines in the cluster as load generators. The load generators use an open loop that generates UDP packets at a fixed rate and consumes their echoes when they arrive. The payload of each packet is a random, but fixed string of 1024 bytes. We configure the load generators to generate load that yields the best attainable throughput for each system and

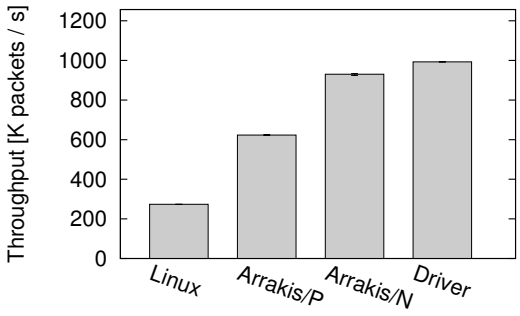

Figure 3. Average UDP echo throughput for packets with 1024 byte payload. The top Y axis value shows theoretical maximum throughput on the 10G network. Error bars show min/max measured over 5 repeats of the experiment.

run each experiment at this load for 20 seconds. Table 1 of Section 2.1 shows a breakdown of the average server-side latency involved in processing each packet.

Due to the reduced code complexity of an application-level network stack and the removal of the kernel from the data plane, Arrakis requires uniformly lower processing overhead than Linux. By relying on the hardware for packet demultiplexing and a dedicated, application-level network stack, we are able to eliminate 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 1.44 us, 1.92 us (57%) less than that of Linux. 0.85 us (59%) of Arrakis's overhead is due to copying packet buffers to adhere to the POSIX interface. When using Arrakis's native interface (Arrakis/N) that eliminates all copying, we are able to reduce processing overhead to 0.41 us (which is 28% that of Arrakis/P). The additional 13% savings are due to the elimination of additional code to determine total packet size and reduced cache pollution due to the omission of the data copies.

Figure 3 shows the average throughput attained by each system over several repeats of the UDP echo benchmark. At 623,246 packets/s, Arrakis/P achieves 2.28x the throughput of Linux (273,478 packets/s). By departing from POSIX, Arrakis/N achieves a throughput of 930,579 packets/s, 1.9x that of Arrakis/P, or a total throughput of 3.4x that of Linux.

To gauge the maximum possible throughput, we embedded a minimal echo server directly into the NIC device driver, eliminating any remaining API overhead. The device driver achieves a throughput of 992,817 packets/s, 85% of the theoretical line rate (1,170,000 packets/s). Note that both the Arrakis/N and driver embedded versions still read each packet entirely before echoing it back.

4.2 Cloud Infrastructure Application Performance

The microbenchmarks highlight the best attainable throughput result for each system, by focusing all computation on network packet processing. We now turn our attention to the performance achieved by applications, which take processor time away from network packet processing and thus make it a less prominent part of the overall cost. Will we still see significant improvements?

Many cloud infrastructure workloads consist of a large number of short transactions. We have an interest in processing each transaction as quickly as possible as this means a larger number of users can benefit from these services. It can also shorten the latency of a higher-level user request that might result in many requests to backend infrastructure services. For example, to return a cached entry from the memcached distributed object caching system, a request-response transaction over the network is required. Likewise, HTTP load balancers and web proxy servers have to deal with high volumes of individual client HTTP transactions. A high-level user request to a web service operating on cloud infrastructure might result in several backend requests to each of these services.

In this subsection, we investigate the average achieved transaction throughput of two of these cloud infrastructure workloads:

- A typical load pattern observed in many large deployments of the memcached distributed object caching system, and
- a workload consisting of a large number of individual client HTTP requests made to a farm of webservers via an HTTP load balancer.

4.2.1 Memcached

Many cloud applications employ distributed object storage systems, such as memcached.[2] Memcached is an in-memory object caching system that can be used to store frequently accessed web objects and allows them to be fetched at much lower latencies than by requesting them from a database or a file service. Memcached deployments are often hit by large amounts of object requests by frontend servers and transaction throughput is an important metric of the effectiveness of such a deployment.

We observed that the memcached program incurs a processing overhead between 2 and 3 us on our systems for an average object fetch request. This is low enough to merit an investigation into whether reducing OS network processing overhead would benefit the throughput of the deployment.

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 unchanged.

Typical deployments of memcached use pre-arranged TCP connections or connectionless UDP to facilitate the

[2] `http://www.memcached.org/`.

processing of large numbers of transactions. Researchers have reported that real deployments of memcached encounter workloads where the amount of object fetches is much greater than the amount of object store requests [3].

We replicate this setup by installing memcached version 1.4.15 on the server machine for each of our systems and use the other five machines as load generating clients. We implemented our own workload generator program that uses memcached's binary UDP protocol and sends requests at a configurable rate in an open loop. We verified that this achieves similar throughput to that measured using the popular memslap[3] benchmark, which uses a closed benchmarking loop.

We configure a workload pattern of 90% object fetches and 10% object store requests on a pre-generated range of 128 different keys of a fixed size of 64 bytes and a value size of 1024 bytes. We determine at which request rate each system achieves the highest throughput and run the following benchmark at the optimal request rate for each system for 20 seconds.

To measure network stack scalability with number of CPU cores, we repeat the benchmark as we increase the number of memcached server processes executing on the server. Each server process executes independently on its own port number, such that measurements are not impacted by potential scalability bottlenecks in memcached itself. We configure the load generators to distribute load equally among the available memcached instances. On Linux, memcached processes still 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 Arranet.

We show the results in Figure 4 and observe that memcached, when executing on Arrakis/P, achieves a throughput 1.7x that of the same version executing on Linux. Increasing the number of memcached instances on Arrakis scales linearly with an increasing number of CPU cores and attains near line-rate (top of graph) at four CPU cores. Throughput drops slightly when moving to a six core setup. This is due to background system management processes executing on the last CPU that cannot be stopped and take processing time away from the memcached instance running on that core.

Linux scales at a factor of 1.7x for the first doubling of CPUs and is now at half the throughput of Arrakis/P. It then plateaus to a throughput increase of 1.2x per the next doubling of CPUs, while Arrakis continues to scale.

We also measure the throughput of a single memcached instance using threads instead of processes, but observe no noticeable difference to the multi-process scenario. This is not surprising as memcached is optimized to scale well with an increasing number of CPUs and utilizes only one fine-grained lock around its hashtable data structure.

[3] `http://www.libmemcached.org/`.

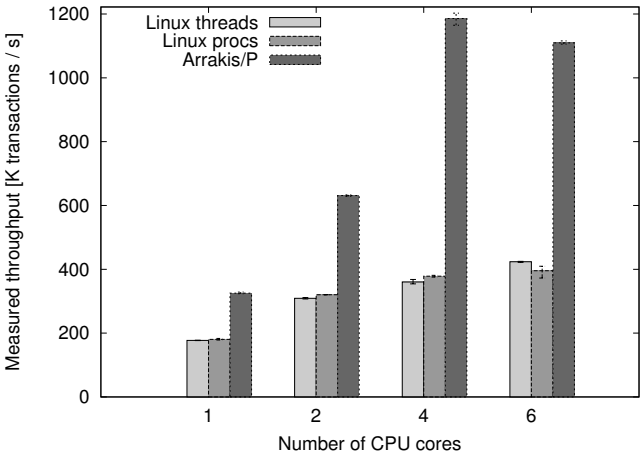

Figure 4. Average memcached transaction throughput when executing on different numbers of CPU cores. Error bars show min/max measured over 5 runs of the experiment. The top y-axis shows the theoretical maximum throughput achievable in this experiment at a speed of 10Gbits/s.

We conclude that, even with the additional processing overhead of memcached, the control-plane/data-plane split of Arrakis still allows for substantial application-level throughput improvements when compared to Linux, which involves the kernel for even the data plane operations. Note however that since the experiment is carried out on an isolated network with a single connectionless socket, Linux does not incur the cost of additional system calls to create and destroy individual connections. Still, its performance is reduced due to the overheads of its network stack.

We note that memcached is an excellent example of the communication endpoint abstraction: We can create hardware filters to allow packet reception and transmission only between the memcached server and a designated list of client machines that are part of the cloud application. In the Linux case, we either have to employ individual connections or use connectionless UDP, which receives packets from arbitrary sources and then filter them in the application.

Turning to scalability, the tight separation of kernel-level network stack and user-space application in Linux means that the network stack cannot simply assign incoming packet flows to hardware device queues to be handled directly by application threads. It also has only limited information about which application threads might be responsible for packet processing and hence has difficulty assigning threads to the right CPU cores that have the packet data hot in their cache. The shared nature of the Linux socket data structures also means that locks need to be taken out when socket queues are accessed. The resulting cache misses and lock contention are responsible for the rest of the overhead.

In Arrakis, the application is in control of the whole packet processing flow: assignment of packet flows to packet queues, packet queues to cores, and finally the scheduling of its own threads on these cores. The application can thus arrange

for packets to always arrive on a queue assigned to a core executing the packet handling threads. The network stack thus does not need to acquire any locks, and packet data is always available in the right processor cache.

4.2.2 HTTP Load Balancer

To aid scalability of web services, HTTP load balancers are often deployed to distribute client load over a number of frontend web servers. A key performance figure for HTTP load balancers is the number of HTTP transactions they are able to sustain without becoming a performance bottleneck themselves.

A popular HTTP load balancer employed by many web and cloud services is haproxy.[4] haproxy is used by web sites with high client load, such as GitHub.com, Instagram, and Twitter. It is also frequently deployed in the Amazon EC2 cloud. 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.

To investigate how performance is impacted when many connections need to be maintained, we conduct an experiment to examine the throughput limit of HTTP transactions on haproxy. We replicate a cloud deployment with five web servers and one load balancer on our cluster. To minimize overhead at the web servers, we deploy a simple static web page of 1,024 bytes, which is served by haproxy out of main memory. We found this solution to yield the highest per-server throughput, compared to other web server packages such as Apache and lighttpd.

We use these same web server hosts as workload generators, using ApacheBench version 2.3. We found this setup to yield better throughput results than other partitions of client/server duties among the cluster nodes. On the load balancer host, we deploy haproxy version 1.4.24, which we configure to distribute incoming load in a round-robin fashion among the web servers.

We configure the ApacheBench instances to conduct as many concurrent requests for the static web page as possible, opening as many concurrent connections to the single haproxy instance as is necessary. We execute the benchmark in this way for 20 seconds. Each request is encapsulated in its own TCP connection. haproxy relies on cookies, which it inserts into the HTTP stream of each connection, after determining assignment of incoming new connection requests to web servers. It uses these cookies to remember its assignment, despite possible client re-connects. This requires it to investigate the HTTP stream at least once for each new 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 the Linux operating system when it decides that this is beneficial.

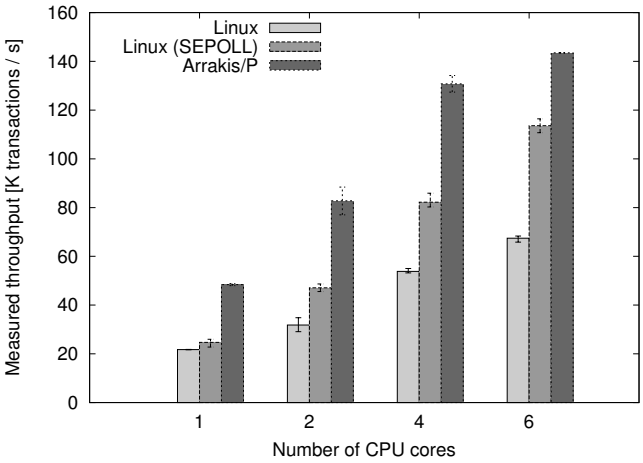

Figure 5. Average HTTP transaction throughput on haproxy when executing on different numbers of CPU cores. Error bars show min/max measured over 5 runs of the experiment.

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 Arranet 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.

To conduct the scalability benchmark, we run multiple copies of the haproxy process on the load balancing node, each executing on their own port number. We configure the ApacheBench instances to distribute their load equally among the available haproxy instances.

In Figure 5, we can see that Arrakis outperforms Linux in both regular and SEPOLL configurations, by a factor of 2.2x and 2x, respectively. Both systems show equivalent scalability curves. We expect the slightly less perfect scalability to be due to the additional overhead induced by TCP connection handling (SYN, ACK and FIN packets) that is not included in the figure. Unfortunately, we do not own a multi-core machine large enough to investigate where we hit a scalability limit. The lower scalability factor in Arrakis's case on 6 CPUs is again due to the background activity on the last CPU.

To conclude, connection oriented workloads require a higher number of system calls for maintenance operations, such as setup (the **accept** system call and possible invocations of **setsockopt** to configure the new socket) and teardown (the **close** system call). 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

[4] `http://haproxy.1wt.eu`.

service. Further socket operations are reduced to function calls in the library OS, which have lower overhead.

4.3 Arrakis Native Interface Case Study

Table 1 shows that, by using the native Arrakis API described in Section 3.4, we can eliminate 60% of the network stack overhead present in Arrakis/P.

By returning a pointer to the payload of a received packet, instead of requiring the user to specify its own buffer space, the native API allows the application to interact directly on packet data provided by the NIC on the receive side, eliminating a buffer copy that is required in the POSIX API. Likewise, by providing a scatter/gather I/O interface and an asynchronous completion notification on the transmit interface, the native API saves a buffer copy, which would be required for performance even with the **sendmsg** POSIX API call to allow buffering of packet data. The POSIX interface requires synchronous completion of the I/O upon return from the call and the application is free to overwrite any specified data after completion.

As a case study, we modified memcached to make use of the native Arrakis interface, both on the receive and on the transmit side. Only limited changes needed to be made to support the new interfaces. In total, 74 lines of code were changed, with 11 pertaining to the receive side, and 63 to the send side interface.

On the receive side, the changes involve eliminating memcached's receive buffer and working directly with pointers to packet buffers provided by Arranet, as well as returning each buffer to Arranet when receive-side processing is completed. The changes amount to an average performance increase of 9% in the memcached benchmark from Section 4.2.1.

On the send side, the changes involve the allocation of a number of send buffers to allow buffering of responses until fully sent by the NIC, which now has to be done within the memcached application. They also involve the addition of reference counts to hashtable entries and send buffers to determine when it is safe to free or overwrite buffers and hashtable entries that might otherwise still be processed by the NIC. We gain an additional average performance increase of 10% when using the send side API in addition to the receive side API.

With these modest changes to make use of the native Arrakis API, memcached is able to achieve a total performance increase of 19% over Arrakis/P. While evaluating our changes, we found that it is important to be careful about proper memory alignment of packet payload and reading the packet data to the appropriate CPU cache within the Arranet network stack. For example, using the native interface, memcached's hash function operates directly on the payload of the received packet. The hash function is very sensitive to proper data alignment and a slowdown due to misalignment can easily nullify any performance improvements gained by eliminating a data copy. The 82599 provides a hardware feature to automatically split packet headers from payload and store each in a different receive buffer, which helps aligning the packet payload.

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.

5. Discussion

In this section, we discuss how we can extend the Arrakis model to apply to solid state and magnetic disk operations, as well as to interprocessor interrupts.

5.1 Virtualized Disk I/O

Database designers have long recognized the performance and scalability limitations of the traditional operating system architecture for I/O-intensive applications [27]. Although—or perhaps because—file systems are designed to be general-purpose, their semantics are often too weak and their performance too slow for the database to use as a storage manager. Typically, the database knows precisely how data should be arranged on disk, and what constraints must be obeyed in terms of applying updates in order to maintain transactional semantics.

A practical workaround exists for the special case where the database's storage requirements are known in advance, and the database can be assumed to be part of the kernel's trust domain. A portion of the disk is pre-allocated for the database's exclusive use, and the disk device controller is memory-mapped into the database runtime system. The database can then schedule reads and writes onto its portion of the disk without mediation by the kernel. Any other application interested in reading data from that portion of the disk is required to go through the database in order to retrieve it.

Our approach is a generalization of this idea to work with multiple applications. Although our approach requires hardware changes in the disk controller, we observe that commercial disk hardware already has much of what is needed. Modern magnetic and solid state disk devices insert a block remapping layer for handling bad block avoidance and wear-levelling. We likewise need a block remapping layer to translate from application-level block numbers to physical block numbers. Because disks need to handle many concurrent operations to get good disk scheduling or wear-levelling performance, and those operations can complete out of order, disk device controllers already have a request and completed operation queue quite similar to the asynchronous transmit and receive queues found in network cards. To this

we would need to add the fault isolation and resource limits discussed in Section 3.

Our model is that most applications will make use of a kernel-resident file system, as is the case today. However, for performance or reliability sensitive applications, we provide the ability for those applications to allocate chunks of disk space, and then manage those chunks directly from a user-level library without kernel intervention. These chunks can be thought of as equivalent to LFS segments [25] – large enough to prorate the disk seek in the common case that the application accesses the chunk sequentially. Unlike LFS, the application has control over the segment layout and write order, allowing individual applications to choose journalling, LFS, or WAFL depending on application needs. For example, a web cache might format its data using WAFL, while a video editor might use journalling. The disk controller enforces access control, translating a segment:offset to a physical disk block. The kernel-resident file system operates on this same interface, allocating space as needed, one chunk at a time.

As in the database workaround we described above, any other applications that need access to file data (e.g., for backup or local search) must retrieve it from the application that wrote it. A file system lookup proceeds as in a traditional operating system, except when the lookup reaches a part of the name space managed by an application (e.g., the web cache), the application is invoked with an NFS-style RPC protocol to return the linearized file data.

5.2 Virtualized Interprocessor Interrupts

To date, most parallel applications are designed assuming that shared-memory is (relatively) efficient, while interprocessor signalling 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 signalling an event between two threads running inside the same application.

Without kernel mediation, 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 signalling among cores running the same application, and to require kernel mediation between cores running different applications.

6. Related Work

Previous work on reducing kernel overheads for I/O operations has focused on minimizing the amount of kernel-level processing for each operation, although not to the extreme degree as Arrakis. SPIN [9] and Exokernel [13] both reduced shared kernel components to allow each application to have customized operating system management. Exokernel, in particular, targets the overheads introduced by the operating system in I/O operations. Nemesis [10] also reduces shared components to provide more performance isolation for multimedia applications. Arrakis is able to completely eliminate kernel-level processing for normal I/O operations providing all of the benefits of having customized I/O stacks for each application.

Other work has focused on more general inefficiencies in the operating systems networking stack. Affinity-Accept [22] identifies and proposes a solution for the misdirection of I/O interrupts on a multicore system. Arrakis avoids this problem by having the hardware directly deliver interrupts to the application. I/O Lite [21] eliminates redundant copying throughout the network processing stack. We alos found copying to be a major source of overhead in the Linux networking stack, which we were largely able to eliminate in Arrakis.

The High-Performance Computing field has long been interested in transferring data directly between user programs and the network, bypassing the OS which is often viewed as an undesirable source of "noise" [7, 23]. A sequence of hardware standards such as U-Net [28], VIA [11], Infiniband [15] and Remote Direct Memory Access (RDMA) [12] with associated programming models such as Active Messages [29] have addressed the challenge of minimizing, or eliminating entirely, OS involvement in sending and receiving network packets in the common case. However, these new technologies require significant changes to applications and only work in the enclosed environments that HPC applications run in. Arrakis works with unmodified applications, does not require special hardware, and works over wide-area networks.

Arrakis takes advantage of hardware multiplexing introduced for virtualization to completely eliminate the operating system kernel from fast-path operations. In this respect, we follow on from previous work on Dune [8], which used virtualization extensions such as nested paging in a non-virtualized environment, and Exitless IPIs [14], which presented a technique to demultiplex hardware interrupts between virtual machines without mediation from the virtual machine monitor.

7. Conclusion

In this paper, we described and evaluated Arrakis, a new operating system designed to remove the operating system kernel from the I/O data path. Unlike a traditional operating system kernel, 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 for user-level network device access. At a microbenchmark level, we show the potential for an

order of magnitude performance improvement from eliminating unnecessary work: domain crossings, thread context switches, buffer copies, shared kernel locks, and library implementation complexity. We have also measured Arrakis on two widely used network services, a web object cache and an application-level load balancer. In each case, Arrakis was able to significantly outperform native Linux.

References

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

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

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, London, England, UK, 2012.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Oct. 2003.

[5] A. Baumann. Inter-dispatcher communication in barrelfish. Technical report, ETH Zurich, 2011. Barrelfish Technical Note 011.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

[7] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

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

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–283, Copper Mountain, CO, USA, 1995. ACM.

[10] R. Black, P. T. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *Proceedings of the 22nd Annual IEEE Conference on Local Computer Networks*, LCN '97, pages 179–188, 1997.

[11] Compaq Computer Corp., Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification*, version 1.0 edition, December 1997.

[12] R. Consortium. Architectural specifications for RDMA over TCP/IP. `http://www.rdmaconsortium.org/`.

[13] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.*, 20(1):49–83, Feb. 2002.

[14] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: bare-metal performance for i/o virtualization. In *Proceedings of the 17th International conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 411–422, London, England, UK, 2012.

[15] Infiniband Trade Organization. Introduction to Infiniband for end users. `https://cw.infinibandta.org/document/dl/7268`, April 2010.

[16] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet*, December 2010. Revision 2.6.

[17] Intel Corporation. *Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide*, Aug. 2013. Reference Number: 326003-003.

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

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

[20] lwIP. `http://savannah.nongnu.org/projects/lwip/`.

[21] V. S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM Transactions on Computer Systems (TOCS)*, 18(1):37–66, 2000.

[22] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 337–350. ACM, 2012.

[23] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 55–, Phoenix, AZ, USA, 2003.

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

[25] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[26] Solarflare Communications, Inc. *Solarflare SFN5122F Dual-Port 10GbE Enterprise Server Adapter*, 2010.

[27] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, July 1981.

[28] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating*

Systems Principles (SOSP '95), Copper Mountain, CO, USA, 1995.

[29] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, Queensland, Australia, 1992.