Arrakis: The Operating System is the Control Plane

Simon Peter   Jialin Li   Irene Zhang   Dan R.K. Ports   Timothy Roscoe
Doug Woos   Arvind Krishnamurthy   Thomas Anderson   *ETH Zurich*
University of Washington

Abstract

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 2-5x end-to-end latency and 9x throughput improvements for a popular persistent NoSQL store relative to a well-tuned Linux implementation.

1 Introduction

Reducing the overhead of the operating system process abstraction has been a longstanding goal of systems design. This issue has become particularly salient with modern client-server computing. Many servers spend much of their time executing operating system code: delivering interrupts, demultiplexing and copying network packets, and maintaining filesystem meta-data. 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. Further, the combination of high speed Ethernet and low latency persistent memories are widening the gap between what is possible running in kernel mode and what is available to applications.

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 [42], reducing the overhead for large numbers of connections [26], protocol specialization [40], resource containers [9, 36], direct transfers between disk and network buffers [42], interrupt steering [43], hardware TCP acceleration, etc. Much of this has been adopted in mainline commercial OSes, and 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 [18, 21, 50]. Although commercially unsuccessful at the time, the virtualization market has now led hardware vendors to revive the idea [6, 35, 44], and also extend it to disks [48, 49].

This paper explores the OS implications of removing the kernel from the data path for nearly all I/O operations. 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, e.g., by allowing applications unfiltered direct access to the network or the 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 2x better read latency, 5x better write latency, and 9x 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 (§3).

- We implement prototypes of our model as a set of modifications to the open source Barrelfish operating system, running on commercially available multi-core computers and I/O device hardware (§3.6).

- We use these prototypes to quantify the potential benefits of user-level I/O for several widely used network services, including a key-value store, a NoSQL store, an IP-layer middlebox, and an HTTP load balancer (§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 (§4.3).
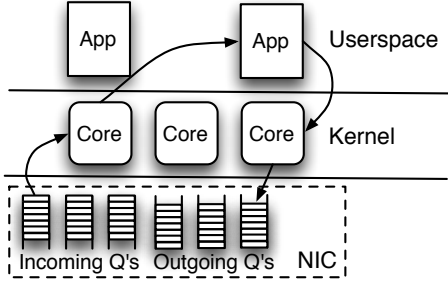
1

Figure 1: Linux networking architecture and workflow.

Figure 2: Overhead in μs of various Linux filesystem implementations, when conducting small, persistent writes.

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 6-core Intel Xeon E5-2430 (Sandy Bridge) systems at 2.2 GHz clock frequency executing Ubuntu Linux 13.04, with 1.5 MB L2 cache and 15 MB L3 cache, 4 GB of memory, an Intel X520 (82599-based) 10Gb Ethernet adapter, and an Intel MegaRAID RS3DC040 RAID controller with 1GB cache of flash-backed DRAM, exposing a 100GB Intel DC S3700 series SSD as one logical disk. 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.

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 1 shows, operating system overhead for packet processing falls into four main categories.

- **Network stack** 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 3.36 μs (see Table 1) spent processing each packet in Linux, nearly 70% is spent in the network stack. This work is mostly 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.
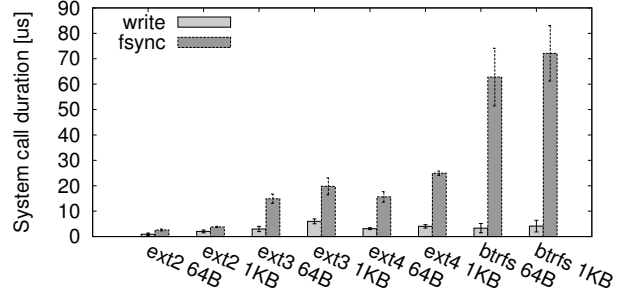
Scheduler overhead depends significantly on whether the receiving process is currently running. If it is, only 5% of processing time is spent in the scheduler; if it is not, the time to context-switch to the server process from the idle process adds an extra 2.2 μs and a further 0.6 μs slowdown in other 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 (aRFS) [4] aims to mitigate this cost, but these solutions themselves impose non-trivial setup costs [43].

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 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 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 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

		Linux				Arrakis			
		Receiver running		CPU idle		POSIX interface		Native interface	
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	($\sigma = 0.66$)	6.19	($\sigma = 0.82$)	1.44	($\sigma < 0.01$)	0.38	($\sigma < 0.01$)

Table 1: 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 currently supports polling only. Thus, the CPU idle case does not apply.

researchers have investigated ways to eliminate this copy from kernel network stacks [42], as Table 1 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[1] system call in a tight loop of 10,000 iterations, measuring each operation's execution 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), as well as metadata updates, in many cases via a journal[49].

Considering both the near-ubiquity of caching RAID controllers and NVRAM-based storage adapters, OS storage stack overhead becomes a major factor. We measured average write latency to our RAID cache to be 25 μs and PCIe-attached flash storage adapters, like Fusion-IO's ioDrive2, report hardware access latencies as low as 15 μs [23]. In comparison, OS storage stack overheads are high, between 40% and 2x for the extended filesystems, depending on journal use, and up to 5x for btrfs.

2.3 Application Overheads

What do these I/O stack overheads mean to operation latencies within a typical datacenter application? Consider the Redis [17] NoSQL store. Redis persists each write via

an operational log (called *append-only file*)[2] 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, 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 minor amounts of time in accounting, access checks, replication (disabled in this experiment), and connection handling (**Other** row in Table 2).

Table 2 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%. Similarly, 90% of the latency of an average 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 is due to the better cache behavior of the user-level I/O stacks and the control/data plane separation evading all kernel crossings. Arrakis' write latency is still dominated by storage access latency (25μs in our system). We expect the latency to improve as faster storage devices appear on the market.

2.4 Hardware I/O Virtualization

Single-Root I/O Virtualization (SR-IOV) [35] 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 resembles

---

[1] We also tried fdatasync, with negligible difference in latency.

[2] Redis also supports snapshot persistence because of the high per-operation overhead imposed by Linux.

| | 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%)
send()	3.17	(36.56%)	0.71	(17.44%)	5.06	(3.10%)	0.33	(1.05%)
Parse input	0.85	(9.80%)	0.66	(16.22%)	2.34	(1.43%)	1.19	(3.78%)
Lookup/set key	0.10	(1.15%)	0.10	(2.46%)	1.03	(0.63%)	0.43	(1.36%)
Prepare response	0.60	(6.92%)	0.64	(15.72%)	0.59	(0.36%)	0.10	(0.32%)
Log marshaling	-		-		3.64	(2.23%)	2.43	(7.71%)
Write log	-		-		6.33	(3.88%)	0.10	(0.32%)
Persistence	-		-		137.84	(84.49%)	24.26	(76.99%)
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	$(\sigma = 1.91)$

Table 2: Overheads in the Redis NoSQL store for memory reads (hits) and durable writes (legend in Table 1).

a PCI device, which can be directly mapped into a different virtual machine and access can be protected via an IOMMU (e.g. Intel's VT-d [31]). To the guest operating system, each virtual function can be programmed as if it was a regular physical device, with a normal device driver and an unchanged I/O stack. Hypervisor software with access to the physical hardware (such as Domain 0 in a Xen [10] 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.

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 which was implemented twenty years ago with U-Net [50], 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 captures the important features of SR-IOV adapters [28, 37, 38, 47]; 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[44]. 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.

While RDMA provides the performance benefits of user-level networking to parallel applications, it is challenging to apply the model to a broader class of client-server applications [20]. Most importantly, RDMA is point-to-point. Each participant receives an authenticator providing it permission to 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. Therefore we do not 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 to take advantage of the Arrakis native interface.

- **Appropriate OS/hardware abstractions:** Arrakis's abstractions should be sufficiently flexible to efficiently support a broad range of I/O patterns, scale well on multi-core systems, and support application requirements for locality and load balancing.

In the rest of this section, we show how we achieve these goals in 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 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 3 shows the overall architecture. In this paper, we focus on hardware that can present multiple instances of itself to the operating system and the ap-
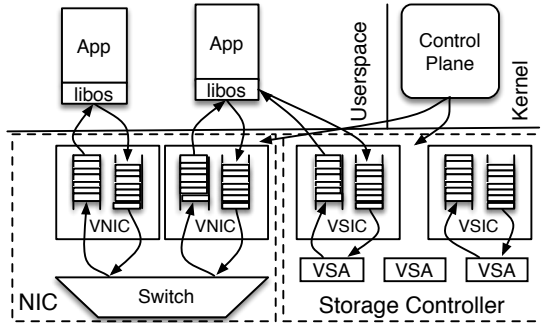
Figure 3: Arrakis architecture. The storage controller maps VSAs to physical storage.

plications running on the node. For each of these virtualized device instances, the underlying physical device provides unique memory mapped register spaces, transmit/receive/command 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 or storage areas, and allocate shared resources to the different instances. Applications conduct I/O through a virtualized device instance without requiring kernel intervention. In order to perform these operations, applications rely on a user-level I/O stack, which is implemented as a library OS. The user-level I/O 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.

The user naming and protection model is unchanged. 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-grain allocation, by associating each application with the directories and files it manages. Other applications can still read those files by indirecting 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) which 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 below.

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 thus further reduces the software I/O overhead.

Transmit and receive filters: A transmit filter is a predicate on network 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. It can also be used to limit an application to only communicate with a pre-selected set of nodes, e.g., for information flow control.

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. 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 kernel control plane.

Virtual storage areas: Storage controllers need to provide an interface via their physical function to create and delete *virtual storage areas* (VSAs), map them to extents of physical drives, and associate them with VSICs. An application can store multiple sub-directories and files in a single VSA, providing precise control over multi-object serialization constraints. A VSA can be dynamically assigned multiple contiguous physical storage extents. On flash, an extent can be an erasure block; on disk, it may be a cylinder group or portion of a cylinder.

Each VSIC may support only one VSA, but in some cases support for multiple different VSAs can have benefits. For example, parallel writes to different storage devices, such as multiple flash chips, improves performance [41]. Existing VSAs may also be extended and shrunk. VSAs are the storage equivalent of NIC filters and ensure access protection without kernel mediation. VSAs can be implemented in hardware via page tables or lists of extents.

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 inter-packet pacing controls associated with the queue. Installation of these controls are also privileged operations.

In addition, we assume that the I/O device driver supports an introspection interface that allows 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., checksum calculations and 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 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 individual bits 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 out of physical disk extents and SSDs use a flash translation layer for wear-leveling. SCSI host-bus adapters support SR-IOV technology for virtualization [37, 38], which allows them to expose multiple VSICs, and the NVMe standard proposes multiple command queues for scalability [32]. Only the required protection mechanism is missing. We anticipate VSAs to be allocated in large chunks (gigabytes) and thus hardware protection mechanisms can be coarse-grained and light-weight.

Finally, the number of hardware-supported VICs might be limited. The 82599 [28] and SAS3008 [38] support 64. This number is adequate with respect to the capabilities of the rest of the hardware (e.g., the number of CPU cores) and 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 2048. To run more applications than the hardware supports, we can simply multiplex the remainder in the traditional way via the kernel to the physical function. Of course, only the first applications would benefit from kernel bypass and this set of high-priority applications would need to be configured by the system administrator.

3.3 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 (subject to resource limitations or policies imposed by Arrakis), and also associate *doorbells* with particular events on particular VICs. A doorbell is an IPC communication endpoint used to notify the application that an event (such as a packet arrival or I/O completion) 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, as well as subsuming 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 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.

As a second example, web servers with a high rate of incoming TCP connections can run into scalability problems processing connection requests [43]. 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 practice this is usually wrapped in a higher-level call **create_filter(flags, peerlist, servicelist) = fil-**

ter 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** 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.

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 and causes Arrakis to configure the underlying hardware accordingly; thereafter that queue can be used to send or receive the corresponding packets.

Similarly, a VSA is acquired via an **acquire_vsa(name) = VSA** call. VSAs are given a global **name**, such that they can be found again when an application is restarted. The returned **VSA** is a capability that allows applications access to the corresponding VSA. The control plane stores capabilities and their association. If a non-existent VSA is acquired, its size is initially zero, but can be resized via the **resize_vsa(VSA, size)** call.

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 each VSA, the control plane maintains a mapping of virtual storage blocks to physical ones and programs the hardware accordingly. As long as 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. Maintaining head locality for hard disks is outside of the scope of our work. 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.

3.4 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. 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 above. 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.

Interaction with the network hardware is designed for maximizing performance from the perspective of both latency and throughput. We 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 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 paper uses function calls to the driver.

Thirdly, 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 which case the 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, as well 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, yield-

ing 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.

3.5 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 (subject to block granularity) 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, queue number, and matching array of ranges (offset and size) within the VSA. The API implementation makes sure to enqueue 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*, TenaciousD, to take advantage of low-latency storage devices. Data structures have the benefit that their storage characteristics can be tailored to their usage model, for improved efficiency, as opposed to a simple read/write interface provided by filesystems. Their drawback is a lack of backwards-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 a persistent log according to these goals and modified Redis to use it (cf. §4.4). The persistent log incorporates metadata directly in the data structure, which allows for early allocation and eliminates any generic metadata book-keeping service, such as a journal. Metadata is allocated once a new log entry is requested in RAM and can thus be persisted while the application fills the entry. The in-memory layout of the log is as stored, eliminating any marshaling that applications typically do to represent the durable version of their data. Furthermore, the log is optimized for linear access and efficiently prefetches entries upon read.

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 from the beginning. The API is asynchronous: an append operation returns immediately and a callback can be registered to be called once it is persistent. This allows us to mask remaining write latencies, e.g., by allowing Redis to optimistically prepare a network response to the client, while the log entry is persisted.

Log entries are allocated in multiples of the storage layer's block size 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 layer, the log ensures that markers are written after the rest of the entry (cf. [16]).

The log itself is identified by a header at the beginning of the VSA that contains an identifier, a version number, the number of log entries, the block size of the storage device, and an offset to the end of the log within the VSA. The log repairs a corrupted or outdated header lazily in the background upon opening, by looking for additional, complete entries from the purported end of the log.

3.6 Implementation

The Arrakis operating system is based upon a fork of the Barrelfish [11] multicore OS code base [1]. We added 33,786 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. We could have also chosen to base Arrakis on the Xen [10] hypervisor or the Intel Data Plane Development Kit (DPDK) [29] 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 [28] 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 [31] and the Intel RS3 family of RAID controllers [30]. 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. 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, Arranet. Arranet 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. Arranet is based in part on the low-level packet processing code of the lwIP network stack [39]. 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 §3.5 and our library of persistent data structures, TenaciousD.

3.7 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-grain filtering in software, within applications. The availability of more general-purpose filters would eliminate this software overhead.

Arranet does not yet support interrupts from the NIC, and all network packet processing is performed 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, Arranet does not notice the arrival and does not allow the application to act immediately. In many settings, server applications receive a continuous stream of requests, using interrupts only as a fallback. Our benchmark applications have this flavor, and so their results are not impacted by this limitation. Nevertheless, support for network interrupts is important future work.

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 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.

4 Evaluation

We evaluate Arrakis on three 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, and 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 an IP-layer middlebox. We also examine the system under maximum load in a series of microbenchmaks 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 §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 non-tuned 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 §2 on the server and use all other machines in the cluster as load generators. The load generators generate 1,024 byte 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 1, 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 1.44 us, 57% less
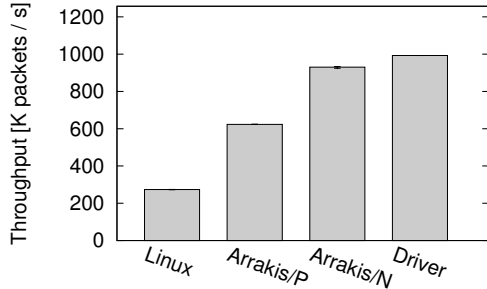
Figure 4: Average UDP echo throughput for packets with 1024 byte payload. The top y-axis value shows theoretical maximum throughput on the 10G network. Error bars show min/max measured over 5 repeats of the experiment.
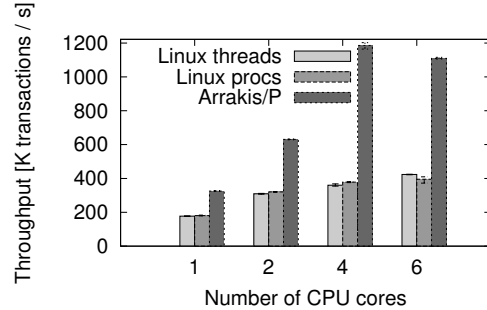

Figure 5: Average memcached transaction throughput and scalability. The top y-axis value shows the maximum throughput achievable in this experiment at 10Gb/s.

than Linux. Arrakis/N is able to reduce this overhead to 0.38 μs.

Figure 4 shows the average throughput attained by each system. Arrakis/P achieves 2.3x the throughput of Linux. By departing from POSIX, Arrakis/N achieves 3.4x the throughput of Linux. Smaller payload sizes showed proportional improvement in throughput on both systems. 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 and achieving 85% of the theoretical line rate (1.17M pps).

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 2–3 μs for an average object fetch request, comparable to the overhead of OS kernel network processing.

We benchmark memcached version 1.4.15 by sending it requests at a constant rate via its binary UDP protocol, using a tool similar to the popular memslap [2] benchmark. We configure a workload pattern of 90% fetch and 10% store requests on a pre-generated range of 128 different keys of a fixed size of 64 bytes and a value size of 1,024 bytes, in line with real cloud deployments [8].

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 Arranet.

Figure 5 shows that memcached on Arrakis/P achieves 1.7x the throughput of Linux on one core, and attains near line-rate at 4 CPU cores. The slightly lower throughput on all 6 cores is due to contention with Barrelfish system management processes [11]. By contrast, Linux

throughput nearly plateaus beyond two cores.

The throughput of a single memcached instance using threads instead of processes shows no noticeable difference to the multi-process 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 CPU cores with packet data hot in their cache. 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, 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 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 Arranet, as well as returning completed buffers to Arranet. The changes amount to an average performance increase of 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
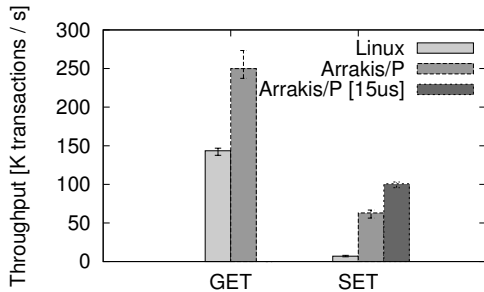
Figure 6: Average Redis transaction throughput.

Figure 7: Average HTTP transaction throughput and scalability of haproxy.

table 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.

4.4 Redis NoSQL Store

Redis [17] extends the memcached model from a cache to a persistent NoSQL object store. Our results in Table 2 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, and a total concurrency of 1,600 connections over 16 benchmark clients executing on the client machines. Redis is single-threaded, so we investigate only single-core performance.

The Arrakis version of Redis is using TenaciousD. We changed 109 lines in the application to manage and exchange records with the TenaciousD log instead of a file. We did not eliminate Redis' marshaling overhead (cf. Table 2). If we did, we would save another 2.43 μs of write latency. Due to the fast I/O stacks, Redis' read performance mirrors that of Memcached and write latency improves by 63%, while write throughput improves vastly, by 9x.

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 backend with 15 μs write latency, such as the ioDrive2 [23]. Write throughput improves by another 1.6x, nearing Linux read throughput.

We conclude that the combination of data-plane network and storage stacks can yield vast benefits in latency and throughput for both read and write-heavy workloads. With a view to upcoming hardware improvements, these benefits will increase further. The tight integration of storage and data structure in TenaciousD allows for a number of latency-saving techniques that eliminate marshaling overhead, book-keeping of journals for filesystem metadata, and can offset storage allocation overhead. Together,
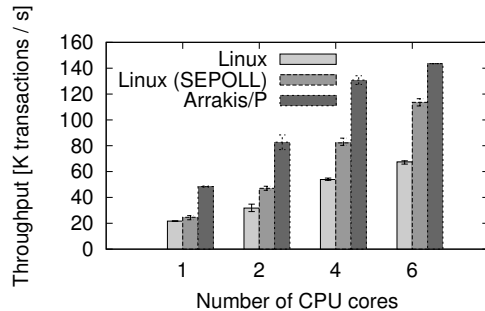
these improvements vastly reduce the software overhead of durable writes.

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 [3]. 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 replicate a cloud deployment with 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, which is served out of main memory.We use these same web server hosts as workload generators, using ApacheBench version 2.3 to conduct as many concurrent requests for the web page as possible. We found this setup to yield better throughput results than other partitions of client/server duties among the cluster nodes. Each request is encapsulated in its own TCP connection. 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. 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.

Haproxy relies on cookies, which it inserts into the HTTP stream to remember connection assignments to backend web servers under 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 Linux when it decides that this is
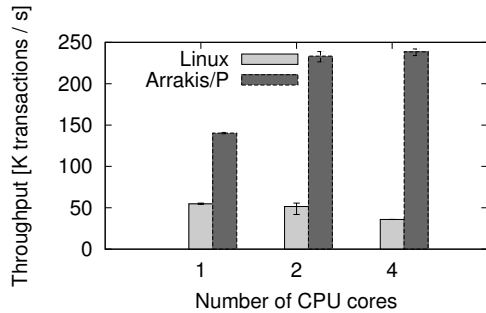
11

Figure 8: Average HTTP transaction throughput and scalability of the load balancing middlebox. Top y-axis value = 10Gb/s.

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 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.

In Figure 7, 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. The slightly less perfect scalability is due to the additional overhead induced by TCP connection handling (SYN, ACK and FIN packets) that is not included in the figure. Note that Arrakis's performance on 6 CPUs is affected by background activity on Barrelfish. Unfortunately, we do not own a multi-core machine large enough to investigate where we hit a scalability limit.

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, which have lower overhead.

4.6 IP-layer Middlebox

IP-layer middleboxes are widespread in wide area networks (WANs) today, especially on the Internet. Common middleboxes perform tasks, such as firewalling, intrusion detection, network address translation, WAN optimization, and load balancing. Due to the complexity of their task, middleboxes can benefit from the programming and run-time convenience allowed by the abstraction, safety, and management capabilities provided by an operating system.

We implemented a simple user-level load balancing middlebox, using raw IP sockets [5]. 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 and without deep knowledge of the higher layer protocols. It simply rewrites source and destination IP addresses and TCP port numbers contained in the packet headers to forward HTTP connections. It uses a hash table to remember existing connection assignments and understands when a TCP connection is active. Responses by the back-end web servers are intercepted yet again and forwarded back to the corresponding clients. This is enough to provide the same load balancing capabilities as in the haproxy experiment. We repeat the experiment from §4.5, replacing haproxy with our middlebox.

The simpler nature of the middlebox can be seen in the throughput results in Figure 8. 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 and Arrakis' benefit is thus more prominent, at 2.6x that of Linux. We also see an interesting effect: the Linux implementation does not scale at all in this configuration. The reason for this are the raw IP sockets, which carry no connection information. Without an indication of which connections to steer to which socket, 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 2 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 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 multi-core parallelism, regardless of protocol specifics.

4.7 Performance Isolation

We show that QoS limits can be enforced in Arrakis, by simulating a simple multi-tenant scenario with 5 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 [27] (specifically, HTB [19]) to rate limit the source port of the equivalent process.

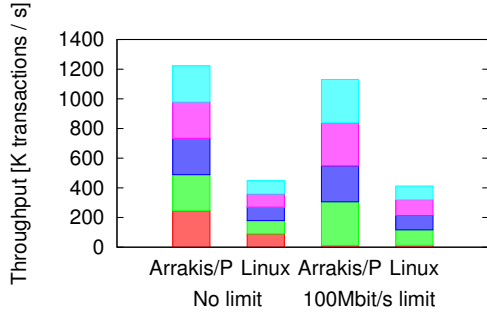We repeat the experiment from §4.2, plotting the throughput achieved by each memcached instance, shown

Figure 9: Memcached transaction throughput over 5 instances (colors), with and without rate limiting.

in Figure 9. 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 5. When rate-limiting, a bit of the total throughput is lost for both OSes 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 Related Work

SPIN [13] and Exokernel [24] reduced shared kernel components to allow each application to have customized operating system management. Nemesis [14] reduces shared components to provide more performance isolation for multimedia applications. Arrakis completely eliminates kernel-level I/O processing, while providing the benefits of customized OS I/O stacks and performance isolation.

We have previously proposed the idea of using hardware virtualization support to eliminate the OS from fastpath I/O operations [7]. In this respect, we follow on from previous work on Dune [12], which used nested paging to provide support for user-level control over virtual memory, and Exitless IPIs [25], which presented a technique to demultiplex hardware interrupts between virtual machines without mediation from the virtual machine monitor.

Netmap [45] implements fast packet-level I/O by doing DMAs directly from user-space packet processors. 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 large numbers of reads and writes. Arrakis, on the other hand, focuses on low-latency I/O. Concurrently with our work, mTCP uses Intel's DPDK interface to implement a scalable user-level TCP [33]; 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 Arranet and mTCP to be similar. OpenOnload [46] 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.

Similarly, recent work has focused on reducing the overheads imposed by traditional filesystems and block device drivers to persistent memory (PM). DFS [34] and PMFS [22] are filesystems 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 PM, avoiding the block layer. Both DFS and PMFS are implemented as kernel-level filesystems, exposing POSIX interfaces. They focus on optimizing filesystem and device driver design for specific technologies, while Arrakis investigates how to allow applications fast, customized device access.

Moneta-D [15] 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 [49] proposes an architecture in which multiple processes communicate with a trusted user-space filesystem service to modify file metadata and perform 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.

6 Conclusion

In this paper, 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 2-5x faster and write throughput 9x higher on Arrakis than on a well-tuned Linux implementation.

References

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

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

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

[4] Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[5] Linux IPv4 raw sockets, May 2012. http://man7.org/linux/man-pages/man7/raw.7.html.

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

[7] Redacted for anonymous reviewing.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[9] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.

[10] 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 *SOSP*, 2003.

[11] 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 *SOSP*, 2009.

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

[13] 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 *SOSP*, 1995.

[14] R. Black, P. T. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *LCN*, 1997.

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

[16] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *SOSP*, 2013.

[17] Citrusbyte. Redis. http://redis.io/.

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

[19] M. Devera. HTB Linux queuing discipline manual – User Guide, May 2002. http://luxik.cdi.cz/~devik/qos/htb/userg.pdf.

[20] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.

[21] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM*, 1994.

[22] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.

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

[24] 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. *TOCS*, 20(1):49–83, Feb 2002.

[25] 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 *ASPLOS*, 2012.

[26] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *OSDI*, 2012.

[27] B. Hubert. Linux advanced routing & traffic control HOWTO. http://www.lartc.org/howto/.

[28] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet*, December 2010. Revision 2.6. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/

```
82599-10-gbe-controller-datasheet.
pdf.
```

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

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

[31] Intel Corporation. Intel virtualization technology for directed I/O architecture specification. Technical Report Order Number: D51397-006, Intel Corporation, Sep 2013.

[32] Intel Corporation. *NVM Express*, revision 1.1a edition, Sep 2013. `http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1a.pdf`.

[33] E. Jeong, S. Woo, M. Jamshed, H. J. S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[34] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, Sep 2010.

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

[36] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun.*, 14(7):1280–1297, Sep 2006.

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

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

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

[40] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *OSDI*, 1996.

[41] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined flash for web-scale internet storage systems. In *ASPLOS*, 2014.

[42] V. S. Pai, P. Druschel, and W. Zwanepoel. IO-Lite: A unified I/O buffering and caching system. In *OSDI*, 1999.

[43] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *EuroSys*, 2012.

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

[45] L. Rizzo. Netmap: A novel framework for fast packet I/O. In *USENIX ATC*, 2012.

[46] SolarFlare Communications, Inc. OpenOnload. `http://www.openonload.org/`.

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

[48] A. Trivedi, P. Stuedi, B. Metzler, R. Pletka, B. G. Fitch, and T. R. Gross. Unified high-performance I/O: One stack to rule them all. In *HotOS*, 2013.

[49] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.

[50] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *SOSP*, 1995.

15