Problem: Building an OS for the Data Center

- Server I/O performance matters:
  - Key-value stores
  - Web & file servers
  - Lock managers
- Can we build an OS that would allow applications deliver performance close to that delivered by data center hardware technology?

*The hardware can help!*

Arrakis Design Goals

- Minimize kernel involvement & deliver I/O directly to applications
  - Reduce OS overhead
- Transparency to the application programmer
  - No requirements for modifications to applications
- Appropriate OS/hardware abstraction
  - Keep classical server OS features
  - I/O protocol flexibility
  - Process protection
  - Global naming

Arrakis Architecture
**Skip the Kernel**

- Kernel
  - API
  - Access Control
  - Copying
  - Global Naming
  - I/O Processing
  - I/O Scheduling
  - Multiplexing
  - Protection
  - Resource Limits
- Redis (Application)
- I/O Devices (Hardware)

**Arrakis Architecture**

- Application (e.g. Redis)
- Kernel
  - API
  - Access Control
  - Copying
  - Global Naming
  - I/O Processing
  - I/O Scheduling
  - Multiplexing
  - Protection
  - Resource Limits
- I/O Devices (Hardware)

**Virtual Interface Card**

- Data path
- User Space
- Hardware Space
- Kernel Space

**Skip the Kernel**

- Kernel mediation is too heavyweight!

**Data Plane**

- Application (Redis)
  - API
  - I/O Processing
- I/O Devices (Hardware)
  - I/O Scheduling
  - Multiplexing
  - Protection

- Copying: A native interface that supports true zero-copy I/O

Source: [https://www.youtube.com/watch?v=4NYpDad0f04](https://www.youtube.com/watch?v=4NYpDad0f04)
Arrakis Control Plane

- Access Control
  - Only do once when configuring the data plane
  - Enforced via NIC filters, logical disks
- Global Naming
  - Virtual file system still in kernel
  - Storage implementation in applications
- Resource Limits
  - Program hardware I/O schedulers

Global Naming

Fig. 6. Arrakis default file access example.
**Arrakis I/O Architecture**

**Control Plane**
- Kernel
  - Access Control
  - Global Naming
  - Resource Limits

**Data Plane**
- Redis (Application)
  - API
  - I/O Processing

**Data Path**
- I/O Devices (Hardware)
  - I/O Scheduling
  - Multiplexing
  - Protection

**Storage Data Plane**

- Persistent Data Structures
  - Examples: persistent log and queue data structures

- Benefits
  - Operations are immediately persistent.
  - The structure is robust versus crash failures.
  - Operations have minimal latency

- Drawbacks
  - A lack of backwards-compatibility to the POSIX API.

**Hardware I/O Virtualization**

- Standard on data center NIC, emerging on RAID

- I/O Scheduling
  - NIC rate limiter, packet schedulers

- Multiplexing
  - Single-Root I/O Virtualization (SR-IOV)
    - Support high-speed I/O for multiple virtual machines sharing a single physical machine.
    - Each virtual PCI device has its own register, queue etc.

- Protection
  - IOMMU
    - Restrict device access to only application virtual memory.
  - Packet filters, logical disks
    - Only allow eligible I/O.
**Evaluation**

- Arrakis was evaluated on four cloud application workloads
  - Read-heavy
  - Write-heavy
  - Http load balancer
  - IP-layer middlebox

- OS configurations used in the evaluation:
  - Ubuntu version 13.04 (kernel version 3.8)
    - Made some tunings and throughput performance improved by 10%
    - Installed latest ixgbe device driver
    - Disabled receive side scaling (RSS) when applications executed on one processor
  - Arrakis using the POSIX interface
  - Arrakis using its native interface

**Server-side Packet Processing Performance**

- **UDP echo server**
  - Other machines generated 1KB UDP packets at a fixed rate for 20 seconds in each experiment
  - the rate at which echoes arrived was recorded and used to compute server-side overhead
  - Arrakis eliminates scheduling and kernel crossing because packets are delivered directly to user space.

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>Arrakis/P</th>
<th>Arrakis/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver run</td>
<td>1.26 (37.9%)</td>
<td>1.24 (20.9%)</td>
<td>0.32 (22.3%)</td>
</tr>
<tr>
<td>CPU in</td>
<td>1.05 (31.3%)</td>
<td>1.42 (22.9%)</td>
<td>0.27 (18.7%)</td>
</tr>
<tr>
<td>CPU out</td>
<td></td>
<td></td>
<td>0.17 (18.0%)</td>
</tr>
<tr>
<td>Scheduler</td>
<td></td>
<td></td>
<td>0.25 (19.9%)</td>
</tr>
<tr>
<td>Copy</td>
<td></td>
<td></td>
<td>0.35 (24.9%)</td>
</tr>
<tr>
<td>Copy out</td>
<td>0.44 (12.3%)</td>
<td>0.55 (8.9%)</td>
<td>0.50 (40.8%)</td>
</tr>
<tr>
<td>Kernel crossing</td>
<td>0.18 (2.9%)</td>
<td>0.20 (3.5%)</td>
<td>-</td>
</tr>
<tr>
<td>syscall</td>
<td>0.03 (2.9%)</td>
<td>0.13 (2.1%)</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>3.36 (σ=0.36)</td>
<td>6.19 (σ=0.82)</td>
<td>1.44 (σ&lt;0.01)</td>
</tr>
</tbody>
</table>

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/P uses the POSIX interface, Arrakis/N uses the native Arrakis interface.

**Read-heavy load**

- Memcached: is a general-purpose distributed memory caching system. It is often used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source (such as a database or API) must be read\[1\].

- **Setup:**
  - Requests were sent at a constant rate via its binary UDP protocol
  - Workload pattern: 90% fetch and 10% store requests
  - Number of Memcached processes were varied to measure network stack scalability for multiple cores

Write-heavy load

- Redis: provides in-memory data structure stores, optionally persists each write via an operational log
  - AOF persistence logs every write operation received by the server
  - RDB persistence performs point-in-time snapshots of dataset at specified intervals [1]
- Log records were exchanged between Redis and Caladan
- Setup:
  - Benchmark tool distributed with Redis
  - Execute GET and SET requests in two separate benchmarks
  - Also ported Caladan to run on Linux
  - Simulated storage hardware with low write latency through a write-delaying RAM disk
- Results:
  - Write latency improves by 63%
  - Write throughput improves by 9X on Arrakis
  - Write throughput improves by 5X on Linux (w/ Caladan)

Http Load Balancer

- Haproxy: high availability proxy, a popular open source software TCP/HTTP load balancer and proxying solution [1]
- Setup:
  - Deployed a static web page of 1024 bytes at five web server, which also served as workload generators
  - Distributed load in a round-robin fashion
  - Experiment was done with and without “speculative epoll” (SEPOLL) within the Linux kernel.
    - SEPOLL: uses knowledge about typical socket operation flows within Linux kernel to avoid calls to the epoll interface and optimize performance
  - Not implemented in Arrakis

IP-layer Middlebox

- IP-layer middleboxes: perform tasks such as firewalling, intrusion detection, network address translation, and load balancing.
- Setup:
  - Implemented a simple user-level load balancing middlebox using raw IP sockets.
    - It simply rewrites source and destination IP addresses and TCP port numbers.
  - A hash table was used to remember existing connection assignment
  - Responses from back-end servers were intercepted and forwarded back to corresponding clients

Results and analysis:
- Load balancing middle box running either Linux or Arrakis experienced a higher throughput compared to Haproxy because of the simpler nature of the middlebox
- Linux implementation does not scale well because raw sockets carry no connection information — each middlebox instance has to look at each incoming packet to determine if it should handle it

Performance Isolation

- Wanted to know if it is possible to provide the same kind of QoS enforcement (rate limiting) in Arrakis as in Linux.
- Setup:
  - Simulated a simple multi-tenant scenario with 5 Memcached instances
    - Limit one tenant’s sending rate to 100Mb/s
  - Used rate specifiers in Arrakis and queuing disciplines on Linux to limit the rate
  - Memcached experiment was repeated
- Conclusion:
  - Arrakis is able to provide the same kind of rate limiting QoS enforcement as in Linux

Discussions

Some applications of Arrakis:

- Make Arrakis as a virtualized guest
  - Moving the control plane into the virtual machine monitor (VMM)
  - Applications allocate virtual interfaces cards directly from VMM
- Virtualized Interprocessor Interrupts
  - Interprocessor signaling is inefficient because of kernel’s involvement even though the sending and receiving threads are two threads of the same application
  - Kernel could be configure to allow an interrupt to be delivered to another processor given that the same application is running on that processor
  - Achieve similar cost as a cache miss

Improvements and Extension

- Throughput of Arrakis does not scale well beyond 4 cores based on the Memcached experiment
  - Reduce overhead caused by contention with Barrellfish system management processes
- Limited filtering support of the 82599 NIC (implementation)
  - Introduce software overhead: different MAC address for each VNIC