An OS for the Data Center

- Server I/O performance matters
  - Key-value stores, web & file servers, lock managers, ...

- Can we deliver performance close to hardware?

- Example system: Dell PowerEdge R520
  - Intel X520 10G NIC
  - Intel RS3 RAID 1GB flash-backed cache
  - Sandy Bridge CPU 6 cores, 2.2 GHz

Today’s I/O devices are fast

Today’s I/O devices are fast

2 us / 1KB packet

25 us / 1KB write

= $1,200
Can’t we just use Linux?
Linux I/O Performance

GET

% OF 1KB REQUEST TIME SPENT

Redis

- HW 18%
- Kernel 62%
- App 20%

9 us

Kernel

- I/O Processing
- Copying
- Protection

Kernel mediation is too heavyweight

Data Path

I/O Processing

10G NIC
2 us / 1KB packet

RAID Storage
25 us / 1KB write
Arrakis Goals

• Skip kernel & deliver I/O directly to applications
  • Reduce OS overhead

• Keep classical server OS features
  • Process protection
  • Resource limits
  • I/O protocol flexibility
  • Global naming

• The hardware can help us...
Hardware I/O Virtualization

- Standard on NIC, emerging on RAID
- Multiplexing
  - **SR-IOV**: Virtual PCI devices with own registers, queues, INTs
- Protection
  - **IOMMU**: Devices use app virtual memory
  - **Packet filters, logical disks**: Only allow eligible I/O
- I/O Scheduling
  - **NIC rate limiter, packet schedulers**
How to skip the kernel?

Redis

Kernel
- API
- Naming
- Access control
- I/O Processing
- Protection
- Multiplexing
- Resource limits
- I/O Scheduling
- Copying

I/O Devices

Data Path
Arrakis I/O Architecture

Control Plane

Kernel
- Naming
- Access control
- Resource limits

Data Plane

Redis
- API
- I/O Processing

I/O Devices
- Protection
- Multiplexing
- I/O Scheduling

Data Path
**Arrakis Control Plane**

- **Access control**
  - Do once when configuring data plane
  - Enforced via NIC filters, logical disks

- **Resource limits**
  - Program hardware I/O schedulers

- **Global naming**
  - Virtual file system still in kernel
  - Storage implementation in applications
Redis Storage Space Allocation

Kernel

create_storage(1G)

Disk space

Free space

Used

Virtual Storage Area

HW ops

create_storage(1G)
Separate Naming From Implementation

Redis → Fast HW ops → Virtual Storage Area

Indirect IPC interface

emacs

Kernel VFS

open("/etc/config.rc")

Logical disk

/tmp/lockfile
/var/lib/key_value.db
/etc/config.rc
...

Kernel VFS
Arrakis I/O Architecture

Control Plane

Kernel
- Naming
- Access control
- Resource limits

Data Plane

Redis
- API
- I/O Processing

I/O Devices
- Protection
- Multiplexing
- I/O Scheduling

Data Path
Storage Data Plane: Persistent Data Structures

• Examples: log, queue
• Operations immediately persistent on disk

Benefits:
• In-memory = on-disk layout
  • Eliminates marshaling
• Metadata in data structure
  • Early allocation
  • Spatial locality
• Data structure specific caching/prefetching

• Modified Redis to use persistent log: 109 LOC changed
Arrakis Device Emulation

Translation Table

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>App 0, VSA 0</td>
<td>Disk 1, LBA 10G, size 1GB</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Round-robin over all apps

Virtual Descriptors

Rewritten Descriptors

Memory Buffer
Evaluation
Redis Latency

• Reduced (in-memory) GET latency by 65%

- Linux:
  - HW 18%
  - Kernel 62%
  - App 20%
  - 9 us

- Arrakis:
  - HW 33%
  - libIO 35%
  - App 32%
  - 4 us

• Reduced (persistent) SET latency by 81%

- Linux (ext4):
  - HW 13%
  - Kernel 84%
  - App 3%
  - 163 us

- Arrakis:
  - HW 77%
  - libIO 7%
  - App 15%
  - 31 us
Redis Throughput

• Improved GET throughput by **1.75x**
  • Linux: **143k** transactions/s
  • Arrakis: **250k** transactions/s

• Improved SET throughput by **9x**
  • Linux: **7k** transactions/s
  • Arrakis: **63k** transactions/s
Redis Throughput

Throughput [k transactions/s]

SET operations

- Linux Intel RS3 [25us] 1x
- Arrakis Intel RS3 [25us] 9x
- Arrakis ioDrive2 [15us]
memcached Scalability

Throughput (k transactions/s)

Number of CPU cores

- 1: 1.8x
- 2: 2x
- 4: 3.1x

10Gb/s interface limit
Getting even more performance…

• POSIX requires data copy for buffering
  • send(): Synchronous packet transmission
  • recv(): User specifies receive location

• Arrakis/Zero Copy
  • Modify send() so that libOS returns buffer when done
  • Modify recv() so that libOS specifies buffer to use

• Port of memcached to Arrakis/Zero Copy
  • TX: 63 LOC changed, 10% better latency
  • RX: 11 LOC changed, 9% better latency
Single-core Performance

UDP echo benchmark

10Gb/s interface limit

Throughput (k packets/s)

- Linux
- Arrakis/POSIX
- Arrakis/Zero-copy
- Driver

- 1x
- 2.3x
- 3.4x
- 3.6x
Implication

We’re all OS developers now.
Future Directions: Devices

• I/O hardware-application co-design
  • At 40 Gbps, even a single cache miss is too expensive

• Application needs fine-grained control (aka OpenFlow)
  • How arriving packets are routed to cores
  • Where in memory or cache to put the packet (or portion of packet)
  • Under control of the sender or receiver, or both

• Similar control needed for persistent memory controllers

• Opportunity to rethink the device driver interface
  • Application-level safe sandboxing of third party drivers
  • Rethink the POSIX API for fast data processing
Future Directions: Storage

• Fast persistent storage is here
  • DRAM+flash, or memristors, or phase change memory

• Rethink distributed systems when networking and persistent memory are both very fast
  • Ex: many data centers observe a non-trivial number of hardware faults
  • On Arrakis, Byzantine fault tolerance protocols that run much faster than today’s Paxos or primary/backup replication

• Application-specific storage system design
  • LFS, WAFL, write-ahead logging, ...
  • Application management of caching, prefetching, and the storage hierarchy
Future Directions: Networking

• Opportunity to rethink congestion control/resource allocation in the data center network
  • TCP mechanics no longer enforced in the OS kernel
  • For multi-gigabit networks, packet loss is a terrible way to signal congestion

• Dynamic negotiation of application-specific network protocols
  • Beyond TCP: PCP, SPDY, QUIC, ...

• Lower OS overhead => more network traffic
  • Network is already a bottleneck
Arrakis Summary

• OS is becoming an I/O bottleneck
  • Globally shared I/O stacks are slow on data path

• **Arrakis**: Split OS into control/data plane
  • Direct application I/O on data path
  • Specialized I/O libraries

• Application-level I/O stacks deliver great performance
  • **Redis**: up to 9x throughput, 81% speedup
  • Memcached scales linearly to 3x throughput

Today’s Data Center Networks
Cost vs. Capacity

• Tension between high cost of network equipment and performance impact of congestion
  • Under-provisioned aggregation/core switches
  • High bandwidth/less congestion within a rack

• Above ToR switches, average link utilization is only 25% at best
• Over a 5 min period, 2.3% of links experience loss

Statistics from Benson ’09 & ’10
Why Is This Happening?

• Rack-level traffic is bursty/long-tailed

This is often a result of **good** job placement, not bad!
Subways

A family of data center architectures that use multiple ports per server
Subways

A family of data center architectures that use multiple ports per server

We do this with edge-only modification and with no additional hardware

• Less traffic in the ToR interconnect
• Remaining traffic is spread more evenly
**Wiring**

- Single ToR per rack
- Shared ToRs w/in a cluster
- Cross-cluster loops

**Load Balancing**

- Uniform random
- Adaptive load balancing
- Detours
### Load Balancing

- **Uniform random**
- **Adaptive load balancing**
- **Detours**

### Wiring

- **Single ToR per rack**
  - Level-0

- **Shared ToRs w/in a cluster**
  - Level-1
    - Level-3
    - Level-5

- **Cross-cluster loops**
  - Level-2
    - Level-4
    - Level-6
Level-1: Shared ToRs w/in a cluster

- Less traffic in the ToR interconnect
- Remaining traffic is spread more evenly
- No changes to routing
Level-2: Cross-cluster Loops

- Load balancing across both racks and clusters
- More shortcuts -> Decreased load on network core
Uniform Random

50%

50% 50%

50%
Adaptive Load Balancing

- Using either MPTCP or Weighted-ECMP
- Better tail latency/less congestion
Detours

- Offload traffic to nearby ToRs
Detours

- Offload traffic to nearby ToRs
- For a single rack, provides full burst bandwidth regardless of oversubscription ratio
Physical Design Considerations
How might we wire a subways loop?
Across Row

Bird’s-eye view
How Might We Wire a Subways Loop?
Evaluation
Improving Memcache Throughput

Max Sustained Items/second

- Level-0: 1x
- Level-1: 1.2x
- Level-3: 1.5x
- Level-5: 2.8x
Faster MapReduce with Less Hardware

![Graph showing completion time vs. ToR oversubscription ratio. The graph indicates a 46% improvement in completion time with less hardware.]
Subways Summary

• Data center network is becoming bottleneck
  • Above ToR, network is *both* congested and under-utilized
• **Subways**: Wire multiple NICs per server into adjacent racks
  • Cross-rack, cross-cluster, aisle-wide dynamic load balancing
• Benefits to application performance/system cost
  • **Memcache**: up to 2.8x better throughput
  • **MapReduce**: equal performance with 1.9x less bandwidth in data center aggregation network
Biography

• College: physics -> psychology -> philosophy
  • Took three CS classes as a senior

• After college: developed an OS for a z80
  • After project shipped, project got cancelled
  • So I applied to grad school; seven out of eight turned me down

• Grad school
  • Learned a lot
  • Dissertation had zero commercial impact for decades

• Post-grad
  • Pick topics where I get to learn a lot
  • Work with people from whom I can learn a lot