Datacenter Operating Systems
CSE451
Simon Peter
With thanks to Timothy Roscoe (ETH Zurich)
Autumn 2015
This Lecture

• What’s a datacenter
  • Why datacenters
  • Types of datacenters

• Hyperscale datacenters
  • Major problem: Server I/O performance

• Arrakis, a datacenter OS
  • Addresses the I/O performance problem (for now)
What’s a Datacenter?

- Large facility to house computer systems
  - 10,000s of machines

- Independently powered
  - Consumes as much power as a small town

- First built in the early 2000s
  - In the wake of the Internet

- Runs a large portion of the digital economy
Why Datacenters?

• **Consolidation**
  • Run many people’s workloads on the same infrastructure
  • Use infrastructure more efficiently (higher utilization)
  • Leverage workload synergies (eg., caching)

• **Virtualization**
  • Build your own private infrastructure quickly and cheaply
  • Move it around anywhere, anytime

• **Automation**
  • No need for expensive, skilled IT workers
  • Expertise is provided by the datacenter vendor
Types of Datacenters

• Supercomputers
  • Compute intensive
  • Scientific computing: weather forecast, simulations, ...

• Hyperscale (this lecture)
  • I/O intensive => Makes for cool OS problems
  • Large-scale web services: Google, Facebook, Twitter, ...

• Cloud
  • Virtualization intensive
  • Everything else: “Smaller” businesses (eg., Netflix)
Hyperscale Datacenters

- **Hyperscale**: Provide services to billions of users
- Users expect response at interactive timescales
  - Within milliseconds
- Examples: Web search, Gmail, Facebook, Twitter

- Built as *multi-tier* application
  - Front end services: Load balancer, web server
  - Back end services: database, locking, replication
- Hundreds of servers contacted for 1 user request
  - Millions of requests per second per server
Hyperscale: I/O Problems

Hardware trend

• Network & storage speeds keep on increasing
  • 10-100 Gb/s Ethernet
  • Flash storage

• CPU frequencies don’t
  • 2-4 GHz

• Example system: Dell PowerEdge R520

- Intel X520 10G NIC
  2 us / 1KB packet

- Intel RS3 RAID
  1GB flash-backed cache
  25 us / 1KB write

- Sandy Bridge CPU
  6 cores, 2.2 GHz
Hyperscale: OS I/O Problems

OS problem

• Traditional OS: Kernel-level I/O processing => slow
  • Shared I/O stack => Complex
  • Layered design => Lots of indirection
  • Lots of copies
Receiving a packet in BSD

Application

Stream socket

TCP
UDP
ICMP

IP

Receive queue

Kernel

Network interface
Receiving a packet in BSD

Kernel

Network interface

Receive queue

Ip

Tcp

Udp

Icmp

Stream socket

Datagram socket

Application

Application

1. Interrupt
   1.1 Allocate mbuf
   1.2 Enqueue packet
   1.3 Post s/w interrupt
Receiving a packet in BSD

1. **Network interface**
   - **Receive queue**
   - **Datagram socket**
   - **Stream socket**

2. **S/W Interrupt**
   - High priority
   - IP processing
   - TCP processing
   - Enqueue on socket

**Kernel**

**Application**

**TCP**  **UDP**  **ICMP**
Receiving a packet in BSD

1. Network interface
2. Receive queue
3. Application
   - Access control
   - Copy mbuf to user space
Sending a packet in BSD

Application → Stream socket
TCP
UDP
ICMP

IP
Receive queue

Kernel

Datagram socket

Network interface
Sending a packet in BSD

1. **Application**
   - Access control
   - Copy from user space to mbuf
   - Call TCP code and process
   - Possible enqueue on socket queue
Sending a packet in BSD

1. Application
2. S/W Interrupt
   Remaining TCP processing
   IP processing
   Enqueue on NIC queue

Kernel

Network interface

Application

Stream socket

Datagram socket

TCP

UDP

ICMP

Receive queue
Sending a packet in BSD

1. Application
2. Stream socket
3. TCP
4. UDP
5. ICMP
6. IP
7. Network interface
8. Receive queue
9. Kernel
10. Application
11. Datagram socket
12. Send packet
13. Free mbuf
14. Interrupt
Linux I/O Performance

<table>
<thead>
<tr>
<th>% OF 1KB REQUEST TIME SPENT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Redis</strong></td>
</tr>
<tr>
<td>GET</td>
</tr>
<tr>
<td>HW 18%</td>
</tr>
<tr>
<td>Kernel 62%</td>
</tr>
<tr>
<td>App 20%</td>
</tr>
<tr>
<td>9 us</td>
</tr>
<tr>
<td>SET</td>
</tr>
<tr>
<td>HW 13%</td>
</tr>
<tr>
<td>Kernel 84%</td>
</tr>
<tr>
<td>App 3%</td>
</tr>
<tr>
<td>163 us</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
</tr>
<tr>
<td>Multiplexing</td>
</tr>
<tr>
<td>Naming</td>
</tr>
<tr>
<td>Resource limits</td>
</tr>
<tr>
<td>Access control</td>
</tr>
<tr>
<td>I/O Scheduling</td>
</tr>
<tr>
<td>I/O Processing</td>
</tr>
<tr>
<td>Copying</td>
</tr>
<tr>
<td>Protection</td>
</tr>
</tbody>
</table>

Data Path

10G NIC
2 us / 1KB packet

RAID Storage
25 us / 1KB write
Arrakis Datacenter OS

• Can we deliver performance closer to hardware?
  • Goal: 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 w/ 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

Library

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 \hspace{1cm} \text{Fast} \hspace{1cm} \text{HW ops} \hspace{1cm} \text{Virtual Storage Area}

Indirect IPC interface

emacs

open("/etc/config.rc")

Kernel VFS

Virtual Storage Area

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

Logical disk

Global Naming
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
**Redis Latency**

- Reduced in-memory GET latency by **65%**

<table>
<thead>
<tr>
<th>Environment</th>
<th>Component</th>
<th>HW</th>
<th>Kernel</th>
<th>App</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>HW</td>
<td>18%</td>
<td>62%</td>
<td>20%</td>
</tr>
<tr>
<td>Arraksis</td>
<td>HW</td>
<td>33%</td>
<td>35%</td>
<td>32%</td>
</tr>
</tbody>
</table>

  - 9 us
  - 4 us

- Reduced persistent SET latency by **81%**

<table>
<thead>
<tr>
<th>Environment</th>
<th>Component</th>
<th>HW</th>
<th>Kernel</th>
<th>App</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (ext4)</td>
<td>HW</td>
<td>13%</td>
<td>84%</td>
<td>3%</td>
</tr>
<tr>
<td>Arraksis</td>
<td>HW</td>
<td>77%</td>
<td>7%</td>
<td>15%</td>
</tr>
</tbody>
</table>

  - 163 us
  - 31 us
Redis Throughput

• Improved GET throughput by \textbf{1.75x}
  • Linux: \textbf{143k} transactions/s
  • Arrakis: \textbf{250k} transactions/s

• Improved SET throughput by \textbf{9x}
  • Linux: \textbf{7k} transactions/s
  • Arrakis: \textbf{63k} transactions/s
memcached Scalability

Throughput (k transactions/s)

<table>
<thead>
<tr>
<th>Number of CPU cores</th>
<th>Linux</th>
<th>Arrakis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2x</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3.1x</td>
</tr>
</tbody>
</table>

10Gb/s interface limit
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
Interested?

• I am recruiting PhD students
• I work at UT Austin

• Apply to UT Austin’s PhD program:
  http://services.cs.utexas.edu/recruit/grad/frontmatter/announcement.html