6.829: Computer Networks

The Role of Operating Systems in Networking

Amy Ousterhout
UC Berkeley

Fall 2019
Why should I care about operating systems?
Loading a Web Page in 1999

- OS overheads: < 2%
Loading a Web Page in 2019

- Significant OS overheads
What Changed?

• Network hardware got faster
• Datacenter apps became more complex
• Lower application latency
  – Store data in memory/flash rather than on disk

![Ethernet Speed Chart](chart.png)

- **Ethernet Speed (Gbits/s)**: 0, 1 Gbits/s, 400 Gbits/s
Clock Speeds Haven’t Kept Up

• Slowing of Moore’s Law
• Operating systems overheads can add significant latency
• CPU efficiency is increasingly important
Load Variation Makes Efficiency Challenging

• Load variation for datacenter workloads
  – Days: diurnal cycles
  – Microseconds: packet bursts, thread bursts
• Peak load requires significantly more cores than average load
• CPUs only utilized 10-66% today
The Rise of Multiplexing

• Two types of applications: latency-sensitive and batch-processing
• Pack both on the same server
  – Bing does this on over 90,000 servers
Multiplexing with Linux

• Example: Memcached + batch processing application
Multiplexing with Linux

- **Linux**:
  - Poor latency
  - Poor throughput

- **Goal**:
  - Some batch throughput

**Diagram Description**:
- The top graph shows the 99.9% latency (µs) against Memcached offered load (million requests/s).
- The bottom graph shows batch ops/s against Memcached offered load (million requests/s).
- The Linux line is represented by a dotted red line, and the Goal line is represented by a solid black line.
## OS Goals for Networked Applications

<table>
<thead>
<tr>
<th>System</th>
<th>Low Tail Latency</th>
<th>Scalable Throughput</th>
<th>High CPU Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>✗</td>
<td>✗</td>
<td>✓ -</td>
</tr>
</tbody>
</table>

overheads of networking and scheduling
How a Packet is Received in Linux

- **Core**
- **User Space**
- **Kernel**
- **NIC Queues**
- **App Task**
- **Interrupt**
- **Load Balance every 4 ms**
- **Socket Buffer**
Linux Overheads

• Network stack adds overhead to every request
  – Layers of queueing
  – Load imbalance
  – Many context switches
  – Complex TCP stack

• Lack of coordination between scheduling and networking
  – Packets can arrive at the wrong core
  – Interference between applications
How Can we Reduce OS Overheads?

• Step 1: bypass the kernel
• Step 2: codesign network stack and CPU scheduling
What is Kernel Bypass?

- Dedicate busy-spinning cores
- Applications directly poll NIC queues
- Enables higher throughput and lower latency

Traditional Approach

Kernel Bypass

```c
sockfd = socket(AF_INET...);
connect(sockfd...);
n = read(sockfd, buf...);
while (1) {
    n = rx_burst(bufs...);
    if (n != 0) ...
}
```
Step 1: Kernel Bypass with IX

- Distribute packets using hash (RSS)
Step 1.5: Handling Load Imbalance with ZygOS

- Static hashes can cause load imbalance
- Balance load with work-stealing
Kernel Bypass is not a Panacea
## OS Goals for Networked Applications

<table>
<thead>
<tr>
<th>System</th>
<th>Low Tail Latency</th>
<th>Scalable Throughput</th>
<th>High CPU Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>✗</td>
<td>✗</td>
<td>✓ -</td>
</tr>
<tr>
<td>ZygOS (SOSP ’17)</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

- **bypass kernel**
- **can’t reallocate cores**
Shenango’s Approach

• CPU scheduling that enables high CPU efficiency and network performance
• Reallocate cores across applications at microsecond granularity
Why Not Millisecond-Scale Reallocations?

- Existing systems reallocate cores every 50-100 ms
  - Can’t react quickly enough to bursts in load

1 μs synthetic work + batch application
Challenges of Fast Reallocations

• How many cores does an application need?
  – Application-level metrics are too slow
  – Multiple sources of load: packets and threads

• Overhead of reallocation
  – Reconfiguring hardware can be slow

• Existing systems don’t address these challenges
Shenango’s Contributions

- Efficient **algorithm** for determining when an application needs more cores
  - Based on thread and packet queueing delays
- **IOKernel**: steers packets in software and allocates cores
- Cache-aware core selection algorithm
- Load balancing of packet protocol (e.g., TCP) handling
Shenango’s Design

App 1
- app thread
- runtime library
- packet queues

App 2
- work stealing

active core

idle core

IOKernel

Kernel

NIC queues

runtime library

packet queues

work stealing
How Many Cores Should the IOKernel Allocate?

1. Packet arrival and no cores
2. Periodic algorithm

App 1
- app thread
- runtime library
- packet queues

active core

idle core

App 2

IOKernel
- runtime library
- packet queues

active core

idle core

Kernel
- NIC queues
What is “Compute Congestion”?  

- Compute congestion: when granting an application an additional core would allow it to complete its work more quickly  
- Goal: grant each application as few cores as possible while avoiding compute congestion
Congestion Detection Algorithm

- Queued threads or packets indicate congestion
- Any packets or threads queued since the last run (5 μs ago)?
  - Grant one more core
- Ring buffers enable an efficient check
  - $\text{head}_{t=n-1} > \text{tail}_{t=n}$ implies congestion
Cache-Aware Core Selection Algorithm

• Grant core with best cache locality

Diagram:
- Warm cache
- Cold cache
- L1 + L2
- 30 MB LLC
- Physical core
- Hyperthreads
Runtime: Load Balancing of Protocol Handling

- Runtime performs protocol handling (e.g., TCP, UDP)
- Work stealing of packets
- Allow packet reordering
  - Resequence packets when necessary
Runtime: C++ Echo Server

```cpp
using namespace rt;

void ServerHandler(void *arg) {
    std::unique_ptr<TcpQueue> q(TcpQueue::Listen({0, server_port}, backlog));

    while (true) {
        TcpConn *c = q->Accept();
        // Thread([=]{ServerWorker(std::unique_ptr<TcpConn>(c));}).Detach();
    }
}

void ServerWorker(std::unique_ptr<TcpConn> c) {
    payload p;

    while (true) {
        // Receive a network request
        ssize_t ret = c->ReadFull(&p, sizeof(p));

        // Send a network response
        ssize_t sret = c->WriteFull(&p, ret);
    }
}
```

- Unlike ZygOS, Shenango has threads!
CPU Efficiency and Network Performance with Memcached

• Shenango matches ZygOS’s tail latency with high CPU efficiency
Shenango is Resilient to Bursts in Load

- TCP requests with 1 μs synthetic work + batch processing application
- Increase or decrease the load every 1 s

![Graph showing Shenango's resilience to bursts in load.](image-url)

- 99.9% Latency (μs)
- Offered Load (million requests/s)
- Time (s)

590 ms

reallocates cores 10,000x as often
The Importance of Congestion Signals

- TCP echo + batch processing application
- Modify congestion detection algorithm
  - Ignore queued threads or queued packets

doesn't work for compute-intensive apps

must dedicate a core

slower reactions
Core Selection Visualization

- Execution trace of which core handles each request
  - Y-axis: CPU core number
  - X-axis: time (1000x slower)
## OS Goals for Networked Applications

<table>
<thead>
<tr>
<th>System</th>
<th>Low Tail Latency</th>
<th>Scalable Throughput</th>
<th>High CPU Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>ZygOS (SOSP ’17)</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Shenango (NSDI ‘19)</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **bottlenecked on the IOKernel**
- **bypass kernel**
- **codesign net + CPU**
Making Shenango Scalable

• Need to move IOKernel off the datapath
• How should we steer packets to cores?
  – Reconfiguring hardware can be slow
Shenango Direct is Scalable

- TCP echo + batch processing application
  - Max throughput with tail latency $< 100 \mu s$
**OS Goals for Networked Applications**

<table>
<thead>
<tr>
<th>System</th>
<th>Low Tail Latency</th>
<th>Scalable Throughput</th>
<th>High CPU Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>✗</td>
<td>✗</td>
<td>✓ -</td>
</tr>
<tr>
<td>ZygOS (SOSP ’17)</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Shenango (NSDI ‘19)</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Shenango Direct</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **bypass kernel**
- **codesign net + CPU**
- **codesign NIC too**
Other OS Challenges for Network-Intensive Applications

• Abstractions
  – What are the right abstractions for interfacing with network hardware?

• Isolation for shared infrastructure
  – How can we isolate applications without degrading performance?
Summary

• Commodity operating systems add significant overheads to network operations
• To reduce these overheads:
  – Bypass the kernel
  – Codesign network stack, NIC, and CPU scheduling
• But… more challenges remain!