Consolidating Workloads at Microsecond Timescales

Adam Belay
abelay@mit.edu

Please do not share outside of Stanford Platform Lab and affiliates
General purpose compute is a bottleneck

[Danowitz, C of ACM 2012]
But significant advances in I/O

<table>
<thead>
<tr>
<th>Year</th>
<th>Networks</th>
<th>Storage</th>
<th>2010</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency:</td>
<td>Latency:</td>
<td>~10 us</td>
<td>~2 us</td>
</tr>
<tr>
<td></td>
<td>Throughput:</td>
<td>Throughput:</td>
<td>10 Gbps</td>
<td>100 Gbps</td>
</tr>
<tr>
<td></td>
<td>75 us</td>
<td>7.5x</td>
<td>35 krps</td>
<td>21x</td>
</tr>
<tr>
<td></td>
<td>10x</td>
<td></td>
<td>35 krps</td>
<td></td>
</tr>
</tbody>
</table>

In 2010:
- Networks: Latency ~10 us, Throughput 10 Gbps
- Storage: Latency 75 us, Throughput 35 krps

In 2020:
- Networks: Latency ~2 us, Throughput 100 Gbps
- Storage: Latency 10 us, Throughput 750 krps
Opportunity: Scaling out

• Can we build larger, more fine-grained distributed systems?
• Complex control flow, resource disaggregation
• Requires OS support for low tail latency and high networking throughput
Solution so far: Kernel-bypass

• Strategy:
  1. Dedicate and pin cores
  2. Busy poll I/O queues

• Research: IX [BPKGKB OSDI 14], Arachne [PLZPWKT OSDI 14], mTCP [JWJJIHP NSDI 14], eRPC [KKA NSDI 19]

• Industry:
“Global data centers used . . . nearly 40% more than the entire United Kingdom. And this consumption will double every four years.”

Forbes, 2017

But busy polling sacrifices efficiency...
Load variation makes efficiency challenging

• Days: Diurnal “day/night” cycles [ADE NSDI 18]
• Hours: Changing load composition [ADE NSDI 18]
• Microseconds: Packet bursts [RZBPS SIGCOMM 15]
• Sharding imbalances; spare capacity for failures

Problem: Peak load requires significantly more cores than average load
Workload consolidation is necessary

• Multiplexing between two classes of apps
  • **Latency-critical**: high priority access to cores
  • **Best-effort**: fills slack cycles

• Keeps CPU load high under bursts and variability
My research

How can OSes maintain microsecond tail latency and high I/O throughput while consolidating several workloads?

- **IX** [OSDI 14] Dataplane OS
- **IXENERGY** [SOCC 15] Power Management
- **SHINJUKU** [NSDI 19] Latency Reduction
- **SHENANGO** [NSDI 19] CPU Multiplexing
- Today’s talk Adversarial Conditions

Special purpose

General purpose
Outline

1. Introduction
2. Microsecond Core Multiplexing
3. Adversarial Conditions
4. Evaluation
5. Conclusion
Shenango’s design

Two components (both run on top of Linux):

1. **Runtimes**: Library OSes linked with each application
   - Flexible, high-level programming abstractions (threads, network sockets, RPC, mutexes, condition variables, etc.)
   - Balances work using **work stealing**

2. **IOKernel**: A centralized core allocator for the entire machine
   - A single, busy-spinning core monitors **queuing delay**
   - Steers packets to active cores and delivers them over shared memory

Tasks migrated in as little as **5 microseconds**, orders of magnitude faster than Linux and other CPU schedulers
Shenango’s system architecture

{Work Stealing}

{Core Allocation}

Key:
- Green: Packet Queues
- Yellow: Thread Queues
IOKernel polls to estimate queuing delay

Queueing delay used as a signal to allocate cores

Key:
- Green: Packet Queues
- Yellow: Thread Queues

IOKernel

CPU 0
Compute congestion control

Adding cores:
• Poll to estimate $D_k$: The queuing delay of each queue k
• $d = \max\{D_k: k = 1, \ldots, n\}$
• If $d > \text{Threshold}$: allocate an additional core

Removing cores:
• Case #1: IOKernel preempts core (if needed by higher priority app)
• Case #2: Runtime runs out of work to steal; voluntarily yields a core
Cores reallocated at microsecond scale

{Work Stealing}

{Core Allocation}

Key:
- Packet Queues
- Thread Queues

IOKernel

CPU 0
Cores reallocated at microsecond scale

{Work Stealing}

{Core Allocation}

Reacts immediately to bursts and changes in load

Key:
- Green: Packet Queues
- Yellow: Thread Queues

IOKernel

CPU 0
IOKernel steers packets to active cores

Optimizes performance by reducing packet stealing
And many more optimizations

1. Cache locality-aware core selection
2. Coordination between runtimes and IOKernel to handle spin locks
3. Efficient, synchronous programming model (like Go)
4. Green thread library (90x faster than Linux, 3x faster than Go)
5. Fine-grained locking and TCP packet resequencing (to handle reordering caused by steering)

See Shenango **NSDI 19** paper for details!
Outline

1. Introduction
2. Microsecond Core Multiplexing
3. Adversarial Conditions
4. Evaluation
5. Conclusion
What about adversarial conditions?

*Shenzhen* improves performance under **worst-case** scenarios

1. **Scalability:** Can we handle many applications and cores? Can we saturate 100 GbE and beyond?

2. **Interference:** Can we mitigate the latency impact of shared CPU resources (hyperthreads, caches, memory bandwidth, etc.)?

3. **Overload:** Can we maintain low tail latency even when a service is under too much load?
Challenge #1: Scalability

The IOKernel is a bottleneck!

• Problem #1: Packet steering overhead
  • Must parse packet headers and deliver to shared memory structures

• Problem #2: Context switching overhead
  • Each preemptive core allocation requires an expensive IPI

IOKernel fails to react quickly when overloaded -> **High Latency**
Idea #1: Offload packet steering (directpath)

{Work Stealing}

Kernel bypass

Update flow steering rules to active cores

 NIC

IOKernel

CPU 0

Key:
- NIC Queues
- Thread Queues

IOKernel can still poll NIC queues to detect delays
Idea #2: Multicast IPIs (ksched kernel module)

![Diagram of idea with CPU and NIC allocation](image)

Key:
- NIC Queues
- Thread Queues

Multiple core preemptions cost about the same as one
TCP networking throughput

Linear scalability with cores, ~10 HT enough to saturate 100GbE
Combined throughput + latency – 10 apps

![Graph showing combined throughput and latency for 10 apps]

**10 apps can achieve same tail latency as one**
Challenge #2: Interference

Cores share CPU resources (cache, memory bandwidth, etc.)

- CPU 0
- CPU 1
- CPU 2
- CPU 3

- L1 + L2
- LLC
- Memory (RAM)

Interference Types:
- Hyperthreading Interference
- LLC Interference
- Memory Bandwidth Interference
### Interference slowdowns

<table>
<thead>
<tr>
<th>Type of interference</th>
<th>Worst-case slowdown</th>
<th>Our mitigation strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperthread</td>
<td>~200%</td>
<td>Prefer self pairings between apps</td>
</tr>
<tr>
<td>LLC</td>
<td>~25%</td>
<td>None (Intel CAT changes take 16 ms)</td>
</tr>
<tr>
<td>Memory BW</td>
<td>~300%</td>
<td>Core throttling</td>
</tr>
</tbody>
</table>

* Note: Linux amplifies these slowdowns because of poor queuing behavior
Our target

Latency impact of memory bandwidth usage

R/W ratios

Better
Low latency requires quick reaction times

Reaction times:
Heracles [ISCA 2015]: 15 s
PARTIES [ASPLOS 2019]: 500 ms
Shenzhen: 5 us (our system)
Core throttling

1: IOKernel polls DRAM BW usage

2: IOKernel samples LLC miss rates

Key:
- NIC Queues
- Thread Queues
Core throttling

1: IOKernel polls DRAM BW usage

2: IOKernel samples LLC miss rates

3: IOKernel takes cores away from BW hog

Key:
- NIC Queues
- Thread Queues

Core Allocation

Work Stealing
Impact of different types of antagonists

- Linux w/ antagonist BE
- Linux w/ mwait BE
- Shenango w/ antagonist BE
- Shenango w/ mwait BE
- Shenzhen w/ antagonist BE
- Shenzhen w/ mwait BE

Hyperthread Interference

LLC Interference

Mem BW Interference

Shenzhen can manage and prevent interference
Impact of changing memory bandwidth

Shenzhen can compensate immediately to bw changes
Challenge #3: Overload

• If request rate exceeds compute capacity, latency grows exponentially
• Basic idea: Drop packets when overloaded (receive livelock)
  • Packet drops cause TCP retransmissions
  • And queuing delays can grow in other places than ingress NIC queue
• Our approach: Use RPC-level congestion control to prevent overload
  • Client stops sending packets when server is busy
  • Goal: Microsecond latency even when highly overloaded
RPC-level congestion control

App 1

{Work Stealing}

CPU 1
CPU 2
CPU 3
CPU 4

Runtime

App 2

Runtime

CPU 5
CPU 6

Key:

NIC Queues
Thread Queues

{Core Allocation}

IOKernel

CPU 0
RPC-level congestion control

1: Suppose app 1 takes all available/provisioned cores

2: IOKernel starts reporting queuing delay directly to App 1’s runtime

Key:
- NIC Queues
- Thread Queues
RPC-level congestion control

Credits issued by server based on queuing delay
Impact of overload

Tail latency stays low when offered load > peak throughput
Outline

1. Introduction
2. Microsecond Core Multiplexing
3. Adversarial Conditions
4. Evaluation
5. Conclusion
Evaluated applications

Latency-critical:
  • Memcached: key-value store (1 us avg. service times)
  • Storage: a flash storage server (10 us avg. service times)
  • Silo: in-memory database (25 us avg. service times)

Best-effort:
  • Streamcluster: A parsec benchmark (hogs all memory bandwidth)

Q: Can we consolidate these applications?
Memcached + Streamcluster

Memcached Latency

Streamcluster Throughput

Better

Linux
Linux (no BE)
Shenzhen
Shenzhen (no BE)
Flash server + Streamcluster

Flash Latency

Streamcluster Throughput

Better

Better
Silo + Streamcluster

Silo Latency

Streamcluster Throughput
All 3 LC apps + 11 x Streamcluster (14 apps)
Outline

1. Introduction
2. Microsecond Core Multiplexing
3. Adversarial Conditions
4. Evaluation
5. Conclusion
Acknowledgements

Amy Ousterhout
Zain Ruan
Joshua Fried
Jonathan Behrens
Aashish Welling
Conclusion

• Low tail latency requires changes at all layers
• But minimizing *queuing delay* is the common theme
• Some challenges remain:
  • NUMA interconnects can get congested
  • IOKernel could be offloaded to NIC hardware to save a core
  • Coordination between network and compute congestion control
• Shenzhen demonstrates microsecond workload consolidation is practical!