Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads
Trend #1: Faster Networks

• But today’s operating systems add significant overheads to I/O
The Rise of Kernel Bypass

- Dedicate busy-spinning cores
- Applications directly poll NIC queues
- Enables higher throughput and lower latency
Trend #2: Slowing of Moore’s Law

- CPUs only utilized 10-66% today
- CPU efficiency becomes 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
The Need for 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 Existing Approaches

- Example: Memcached + batch processing application

$\sim 1 \mu s$
Multiplexing with Existing Approaches

No existing approach provides **high network performance** and **high CPU efficiency**
Goal

• Reconcile the tradeoff between high CPU efficiency and network performance
• Reallocate cores across applications at \textit{microsecond} granularity
  – Coarser granularities insufficient for microsecond-scale tasks and microsecond-scale bursts
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 is too 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
  - Core reallocations take ~5 μs
- Cache-aware core selection algorithm
- Load balancing of packet protocol (e.g., TCP) handling
Shenango’s Design

App 1

App 2

IOKernel

Kernel

work stealing

active core

idle core

app thread

runtime library

packet queues

NIC queues

12
How Many Cores Should the IOKernel Allocate?

- App 1: active core, app thread, runtime library, packet queues
- App 2: idle core
- IOKernel: ② periodic algorithm
- NIC queues
- Kernel: ① packet arrival and no cores
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
Implementation

• IOKernel
  – Uses DPDK 18.11
• Runtime
  – UDP and TCP
  – C++ and Rust bindings
• 13,000 lines of code total
Evaluation Questions

• How well does Shenango reconcile the tradeoff between CPU efficiency and network performance?
• How does Shenango respond to sudden bursts in load?
• How do Shenango’s individual mechanisms contribute to its overall performance?
Experimental Setup

- 1 server + 6 clients, 10 Gbits/s NICs
- Clients run our open-loop load generator built on Shenango
  - Requests follow Poisson arrivals, use TCP

<table>
<thead>
<tr>
<th>System</th>
<th>Kernel Bypass Networking</th>
<th>Lightweight Threading</th>
<th>Balancing Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>✗</td>
<td>✗</td>
<td>4000 μs</td>
</tr>
<tr>
<td>ZygOS (SOSP ’17)</td>
<td>✓</td>
<td>✗</td>
<td>N/A</td>
</tr>
<tr>
<td>Arachne (OSDI ’18)</td>
<td>✗</td>
<td>✓</td>
<td>500000 μs</td>
</tr>
<tr>
<td>Shenango</td>
<td>✓</td>
<td>✓</td>
<td>5 μs</td>
</tr>
</tbody>
</table>
CPU Efficiency and Network Performance with Memcached

- Memcached + batch processing application

- Shenango matches ZygOS’s tail latency with high CPU efficiency

IKernel is saturated

99.9% Latency (μs)

Batch Ops/s

Memcached Offered Load (million requests/s)

kernel bypass networking

no overprovisioning
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 load bursts compared to Arachne with a 590 ms latency and reallocates cores 10,000x as often.](image-url)
Conclusion

• Shenango reconciles the tradeoff between low tail latency and high CPU efficiency

• Reallocates cores at microsecond granularity
  – Efficient congestion detection algorithm
  – IOKernel: allocates cores and steers packets in software

https://github.com/shenango