Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads

Amy Ousterhout  Joshua Fried  Jonathan Behrens  Adam Belay  Hari Balakrishnan
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

Increased demand for servers

Increased demand for energy
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
Multiplexing with Existing Approaches

No existing approach provides **high network performance and high CPU efficiency**
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, threads, storage completions

• 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, packet, and storage completion queueing delays

• **IOKernel**: orchestrates core allocations

• Approach to **dynamic flow steering**
  – Quickly reconfigures NIC steering rules

• Cache-aware core selection algorithm

• Load balancing of packet protocol (e.g., TCP) handling
Shenango’s Design

App 1
- app thread
- runtime library
- NIC queues

work stealing

App 2
- active core
- idle core
- storage queues

IOKernel

13
How Many Cores Should the IOKernel Allocate?

App 1
- app thread
- runtime library
- NIC queues
- storage queues

App 2
- NIC queues

① packet arrival and no cores

IOKernel
- ② periodic algorithm
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
Queued threads, packets, or completions indicate congestion.

- Any items 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

### Congestion Detection Algorithm

1. $\text{head}_t = 0$, $\text{tail}_t = 0$
2. $\text{head}_t = 1$, $\text{tail}_t = 1$

- Congested
- Not congested
Cache-Aware Core Selection Algorithm

- Grant core with best cache locality
Flow Steering

- How should we steer packets to cores?
  - Goals: load balancing and flow affinity

- Asynchronous updates + fast reprogramming mechanisms
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
In this section we evaluate the CPU efficiency and latency of memcached and the spin-server. We used 6 client servers to generate load, enough to minimize client-side queuing delays. We measure each offered load over several seconds, so that bursts come only from TCP, unless stated otherwise.

As shown in Figure 3, Shenango can handle nearly 10 million requests per second. However, we found that Linux's latency degraded significantly due to the presence of background work. This is because at the very low service times of memcached throughputs, both Linux and Arachne were able to scale throughput, both Linux and Arachne were able to scale requests per second with memcached in Linux [50].

2We disabled LRU cache maintenance/eviction and slab rebalancing in Arachne's memcached as the implementation is incomplete.

Similar to previous studies, we achieved 1.6 million requests per second while maintaining a median response time of 25 µs. Despite busy polling on all 16 cores, ZygOS maintains only marginally lower median response time of 70 µs and 99.9th percentile response times (6 and 21 µs less, respectively, at 2 million requests per second). We found that ZygOS can scale to support higher throughput than Shenango, though at a high latency penalty. This is because we achieved 1.6 million requests per second, close to 1.6 million requests per second, with ZygOS.

Arachne2 improved upon Linux, maintaining 99.9th percentile latency below 200 µs. Without background work, Arachne2 improved upon Linux, maintaining 99.9th percentile latency below 200 µs. With ZygOS, we achieved 1.6 million requests per second, though again with milliseconds of tail latency.

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);
    }
}

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();
    }
}

using namespace rt;

Runtime: C++ Echo Server
Implementation

• Runtime
  – UDP and TCP
  – Block storage reads and writes
  – C++ and Rust bindings

• IOKernel
• Ksched kernel module
• 19,000 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 (40 Gbits/s NIC) + 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>50000 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
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 offered load and 99.9% latency](image)

650 ms

reallocates cores 10,000x as often
**Shenango is Scalable**

- TCP echo + batch processing application
  - Max throughput with tail latency < 100 μs

![Graph showing Shenango Direct and Shenango Indirect performance](image-url)
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
Core Selection Visualization

• Execution trace of which core handles each request
  – Y-axis: CPU core number
  – X-axis: time (1000x slower)
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
  – Dynamic flow steering

https://github.com/shenango