Shinjuku: Preemptive Scheduling for Microsecond-Scale Tail Latency

Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, Christos Kozyrakis
Tail latency matters for datacenter workloads

User-perceived latency determined by slowest back-end node

Focus on individual leaf node: minimize tail latency through better scheduling
Achieving low tail latency at microsecond scale is hard

**Problem:** High OS overheads

**Solution:** OS Bypass, polling (no interrupts), run-to-completion (no scheduling)
- Distributed Queues + First Come First Serve scheduling
- **d-FCFS** (DPDK, IX, Arrakis)
Achieving low tail latency at microsecond scale is hard

**Problem:** Queue imbalance because d-FCFS is not work conserving
Achieving low tail latency at microsecond scale is hard

**Problem:** Queue imbalance because d-FCFS is not work conserving

**Solution:** Centralized queue - c-FCFS

Approximation: d-FCFS + stealing e.g., ZygOS
Ideal centralized queue is better in simulation

- d-FCFS: latency starts growing at load ~0.6
- c-FCFS: near optimal performance

Exponential – $\mu = 1\text{us}$

e.g. KVS with homogeneous GET/PUT
Is FCFS good enough when task duration varies?

![Graph showing latency vs load for d-FCFS and c-FCFS.]

- d-FCFS: latency increases even for low load
- c-FCFS: latency increases even for low load

Bimodal – 99.5% 0.5us – 0.5% 500us

e.g. KVS with some RANGE queries

Better
Problem: Short requests get stuck behind long ones

All cores are hogged by long requests
What if we could use the same preemptive scheduling as Linux?

Bimodal – 99.5% 0.5us – 0.5% 500us

PS–1ms: latency increases even for low load (same as c-FCFS)

Bimodal with different latency requirements for low and high load.
Solution: What if we could use preemptive scheduling but at usec scale?

Bimodal – 99.5% 0.5us – 0.5% 500us
e.g. KVS with some RANGE queries

PS-5us: near optimal performance with fast preemption
Insights

Effective scheduling for tail latency requires:

• Centralized queue
• Preemption
• Scheduling policies tailored for each workload

Problem: Microsecond scale requires

• Millions of queue accesses per second
• Preemption as often as every 5us
• Light-weight scheduling policies
**Solution: Shinjuku**

A single address-space operating system that achieves microsecond-scale tail latency for all types of workloads regardless of variability in task duration

**Key Features:**

- Dedicated core for scheduling and queue management
- Leverage hardware support for virtualization for fast preemption
- Very fast context switching in user space
- Match scheduling policy to task distribution and target latency
Outline

• Shinjuku Design
• Preemption Mechanisms
• Scheduling Policies
• Evaluation
Shinjuku Design

1. Process packets and generate application-level requests
2. Pass requests to centralized dispatcher using shared memory
3. Add requests to centralized queue
4. Schedule requests to worker cores using shared memory
5. Send replies back to clients through the networking subsystem
6. Interrupt long running requests and schedule other requests from the queue
Outline

• Shinjuku Design
• Preemption Mechanisms
• Scheduling Policies
• Evaluation
Minimizing Preemption Overhead

Sender Overhead: 2084 cycles
Receiver Overhead: 2523 cycles

- Dispatcher
  - Linux Signal
  - Ring 3 Applications
  - Non-root Ring 0
  - Guest OS
  - Root Ring 0
  - Kernel

- Worker Core
  - Linux Signal
Minimizing Preemption Overhead

Sender Overhead
- 2084 cycles
- 2081 cycles

Receiver Overhead
- 2523 cycles
- 2662 cycles

Dispatcher

Worker Core

Linux Signal
Hardware Interrupts

Ring 3
Applications

Non-root Ring 0
Guest OS

Root Ring 0
Kernel

VMExit

VMExit
Minimizing Preemption Overhead

Sender Overhead
- 2084 cycles
- 2081 cycles
- 298 cycles

Receiver Overhead
- 2523 cycles
- 2662 cycles

Linux Signal
Hardware Interrupts
no VMExits

Dispatcher

Worker Core

Ring 3
Applications

Non-root Ring 0
Guest OS

Root Ring 0
Kernel

Map APIC to dispatcher’s address space

Posted Interrupts

VMExit

VMExit

2084 cycles
-85%
2523 cycles
-52%

2081 cycles
298 cycles

2662 cycles
1212 cycles
Minimizing Preemption Overhead

**Sender Overhead**
- 2084 cycles
- 2081 cycles
- 298 cycles

**Receiver Overhead**
- 2523 cycles
- 2662 cycles

-85% for Sender Overhead
-52% for Receiver Overhead

5us Time Slice

Map APIC to dispatcher’s address space

 Posted Interrupts

**Local APIC**

Root Ring 0
Kernel

Dispatcher

Worker Core

Linux Signal

Hardware Interrupts

no VMExits

298 cycles

5us Time Slice

-85%

2084 cycles

2081 cycles

298 cycles

-52%

2523 cycles

2662 cycles

2081 cycles

2662 cycles

-52%

2084 cycles

2523 cycles

298 cycles

1212 cycles

-85%

2084 cycles

2523 cycles

298 cycles

1212 cycles

-52%

2081 cycles

2662 cycles

298 cycles

1212 cycles

-85%

2081 cycles

2662 cycles

298 cycles

1212 cycles

-52%
Outline

• Shinjuku Design
• Preemption Mechanisms
• Scheduling Policies
• Evaluation
Scheduling policy

1) Which queue to select from?
2) Where to place preempted requests?

Case 1

- Single Queue (SQ)
  - GET
  - PUT
  - GET
  - SCAN

Case 2

- Multiple Queues (MQ)
  - GET
  - PUT
  - PUT
  - GET
  - SCAN
  - SCAN
Queue Selection Policy

**Policy:** Select the queue with the highest ratio: \[
\frac{\text{Waiting Time}}{\text{Target Latency}}
\]

**Short requests:** Initially low Target Latency \(\Rightarrow\) High Ratio

**Long requests:** Eventually high Waiting Time \(\Rightarrow\) High Ratio
Outline

• Shinjuku Design
• Preemption Mechanisms
• Scheduling Policies
• Evaluation
Evaluation

Systems

*Shinjuku* – Centralized preemptive scheduling
- 14 Logical Cores for workers
- 1 Physical Core for both networker and dispatcher (1 Logical Core each)

*IX* – d-FCFS

*ZygOS* – d-FCFS + work stealing
- 16 Logical Cores for workers

Workloads

*Synthetic benchmark* with different service time distributions
*RocksDB* - in-memory database
Shinjuku under low variability

Shinjuku: Close to IX for homogeneous workloads

Synthetic Workload
Exponential – $\mu=1\text{us}$
Shinjuku under high variability

IX and ZygOS: Tail latency determined by SCAN requests

IX and ZygOS: 6.6x faster

RocksDB
99.5% GET - 5us
0.5% SCAN - 250us

88% lower
How important is each optimization?

Single Queue no Preemption

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Latency (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>50%</td>
</tr>
<tr>
<td>SCAN</td>
<td>50%</td>
</tr>
<tr>
<td>GET</td>
<td>50%</td>
</tr>
<tr>
<td>SCAN</td>
<td>1200</td>
</tr>
</tbody>
</table>
How important is each optimization?

**Single Queue with Preemption**

- **50% GET - 5us**
- **50% SCAN - 1200us**

Preemption offers flatter latency for some loss of throughput.

**Shinjuku-SQ no Preemption**

**Shinjuku-SQ with Preemption**
How important is each optimization?

Multiple Queues with Preemption

Multi-queue policy recovers the lost throughput
Does Shinjuku scale?

Synthetic Workload
Fixed 1us

One dispatcher can scale up to 5MRPS and 11 cores
Does Shinjuku scale?

Use multiple dispatchers and scale up to 9.6MRPS
More details in the paper

- Fast context switching
- How Shinjuku supports high line rates
- Placement policy of interrupted requests
- The problems of RSS-only scheduling of requests to cores
- More performance analysis
Conclusion

Low tail latency for general workloads requires:
• Preemptive Scheduling
• Centralized Queueing
• Flexible Scheduling Policies

Shinjuku meets these demands at microsecond scale:
• Scalable centralized queue using dedicated core
• Preemption every 5us
• Latency-driven scheduling policies

github.com/stanford-mast/shinjuku
Backup
Shinjuku Network Scaling

Saturates modern NICs even for small packet sizes
How important is each optimization?

50% GET - 5us

50% SCAN - 1200us

99% GET Latency (us)

99% SCAN Latency (us)

- ZyOS
- Shinjuku-SQ no Preemption
- Shinjuku-SQ
- Shinjuku-MQ
Time slice matters

Slowdown = \frac{Total Latency}{Service Time}

Synthetic Workload
Bimodal
50% 1us – 50% 100us