Problem Overview

The goal is to develop a system that allows the scheduling of mixed latency-critical workloads on a single server while satisfying service level objectives (SLO) and utilizing resources efficiently.
Problem Overview

The goal is to develop a system that allows the scheduling of mixed latency-critical workloads on a single server while satisfying service level objectives (SLO) and utilizing resources efficiently.

App 1 (e.g. key-value store)
SLO: 99% 200us@ 2000QPS

App 2 (e.g. search engine)
SLO: 99% 500us@ 500QPS

CPU
Problem Overview

The goal is to develop a system that allows the scheduling of mixed latency-critical workloads on a single server while satisfying service level objectives (SLO) and utilizing resources efficiently.

App 1a (e.g. KV put/get)
- SLO: 99% 50us
- 10000QPS

App 1b (e.g. KV range query)
- SLO: 99% 500us
- 500QPS
Why build a new system?
Linux Inefficiencies

Linux offers many scheduling options: CFS, RR, BVT but with **HIGH OVERHEADS**
Linux Inefficiencies

Linux offers many scheduling options: CFS, RR, BVT but with **HIGH OVERHEADS**

Bloated Network Stack
Linux Inefficiencies

Linux offers many scheduling options: CFS, RR, BVT but with **HIGH OVERHEADS**.

**Solution:**
Data-plane OS

Bloated Network Stack
Data-plane OS, e.g. IX

NIC

RSS

Packet Queues

Cores
Data-plane OS, e.g. IX

- Minimize communication
- Run-to-completion
- Adaptive batching
But such systems are not work-conserving

- Minimize communication
- Run-to-completion
- Adaptive batching
Work Stealing Can Alleviate The Problem (ZygOS)
Work Stealing Can Alleviate The Problem (ZygOS)
Work Stealing Can Alleviate The Problem (ZygOS)
Work Stealing Can Alleviate The Problem (ZygOS)

- High stealing overhead
- Sub-optimal decisions
- Intra-connection head-of-line blocking
- No preemption

Diagram showing packet queues and cores connected to a NIC, with high steal overhead and sub-optimal decisions indicated.
Shinjuku

1. Centralized Queue
2. Decoupling of Network Processing and Request Execution
3. Scheduling Policies
4. Preemption & Context Switch Mechanisms
Scheduling Policies
Selecting a scheduling policy

- The optimal scheduling policy in terms of tail latency depends on the distribution of the service time of the requests of a workload.
- **Heavy-Tailed** vs **Light-Tailed**
Selecting a scheduling policy

- The optimal scheduling policy in terms of tail latency depends on the distribution of the service time of the requests of a workload
- **Heavy-Tailed vs Light-Tailed** → Processor Sharing & FCFS
Scheduling Approach

- Centralized per request type queues
- Goal → Each request stays within a target latency
Scheduling Approach

1) Which queue to select from?

Request Queues  →  Dispatcher  →  Worker Threads
Scheduling Approach

1) Which queue to select from?
2) How to set a time slice?
Scheduling Approach

1) Which queue to select from?
2) How to set a time slice?
3) Where to enqueue an unfinished request?
Which queue to select from?

Request Queues → Dispatcher → Worker Threads
Which queue to select from?

Idea 1: Queue with the largest normalized backlog
Which queue to select from?

**Idea 1:** Queue with the largest normalized backlog

- Assumes knowledge of the exact execution time of all requests in the system
- Does not take aging into account
Which queue to select from?

**Idea 2:** Check the first request of each queue and select the one with the highest normalized waiting + remaining execution time.
Which queue to select from?

Idea 2: Check the first request of each queue and select the one with the highest normalized waiting + remaining execution time.

(-) Assumes knowledge of the exact execution time of all requests in the system.
Which queue to select from?

**Final Idea:** Check only the normalized waiting time of first request of each queue
How to set the time slice?
How to set the time slice?

Low overheads → **5-15us** Time Slice

Diagram:
- Request Queues
- Dispatcher
- Worker Threads

Diagram labels:
- 2
Where to enqueue an unfinished request?
Where to enqueue an unfinished request?

Idea: Approximate the optimal policy for each request type
Where to enqueue an unfinished request?

Idea: Approximate the optimal policy for each request type

- For **light-tailed** requests enqueue to the **front** of the queue to approximate **FCFS**
Where to enqueue an unfinished request?

**Idea:** Approximate the optimal policy for each request type

- For **light-tailed** requests enqueue to the front of the queue to approximate FCFS
- For **heavy-tailed** requests enqueue to the back of the queue to approximate PS
Preemption & Context Switch Mechanisms

1. Networking subsystem
2. Centralized Dispatcher
3. Worker threads
4. reply
5. Poll-based communication
6. interrupt

Per-core Communication Cache lines

NIC
## Preemption & Context Switch Mechanisms

<table>
<thead>
<tr>
<th>Preemption Mechanism</th>
<th>Sender Overhead (us)</th>
<th>Receiver Overhead (us)</th>
<th>Total Latency (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Signals</td>
<td>0.833</td>
<td>1.009</td>
<td>1.98</td>
</tr>
<tr>
<td>Optimized IPI</td>
<td>0.066</td>
<td>0.485</td>
<td>0.797</td>
</tr>
</tbody>
</table>
## Preemption & Context Switch Mechanisms

<table>
<thead>
<tr>
<th>Preemption Mechanism</th>
<th>Sender Overhead (us)</th>
<th>Receiver Overhead (us)</th>
<th>Total Latency (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Signals</td>
<td>0.833</td>
<td>1.009</td>
<td>1.98</td>
</tr>
<tr>
<td>Optimized IPI</td>
<td>0.066</td>
<td>0.485</td>
<td>0.797</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Overhead (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>swapcontext</td>
<td>0.394</td>
</tr>
<tr>
<td>Optimized Context Switching</td>
<td>0.044</td>
</tr>
</tbody>
</table>
EVALUATION
Evaluation

- Compared **Shinjuku** to **IX** and **ZygOS**
- Synthetic Microbenchmarks with different service time distributions
- RocksDB - in-memory database
- ZygOS, IX: 16 logical worker cores
- Shinjuku:
  - 14 logical worker cores
  - 1 LC for networking subsystem
  - 1 LC for dispatcher
- Optimizing for 99% Latency as function of throughput
Synthetic Workload

(a) Fixed(1)  
(b) Exp(1)  
(c) Single queue  
\text{Bimodal}(99.5 - 0.5, 0.5 - 500)
Synthetic Workload

(a) Fixed(1)  
(b) Exp(1)  
(c) Single queue  
Bimodal(99.5 – 0.5, 0.5 – 500)
Synthetic Workload

Bimodal (50 - 1, 50 - 100) vs Trimodal (33 - 1, 33 - 10, 33 - 100)
Synthetic Workload

Bimodal(50 - 1, 50 - 100)

Trimodal(33 - 1, 33 - 10, 33 - 100)
Importance of Preemption

Bimodal(50 - 1, 50 - 100)
RocksDB

- Mix of GET and SCAN queries
- Need to disable interrupts in “unsafe” sections, e.g. memory allocation and locks
- Configured to be completely in-memory
RocksDB

- Mix of GET and SCAN queries
- Need to disable interrupts in “unsafe” sections, e.g. memory allocation and Locks
- Configured to be completely in-memory

99.5% GET - 0.5% SCAN(1000)
RocksDB

- Mix of GET and SCAN queries
- Need to disable interrupts in “unsafe” sections, e.g. memory allocation and locks
- Configured to be completely in-memory

99.5% GET - 0.5% SCAN(1000)
RocksDB

50% GET - 50% SCAN(5000)
Future Directions

- Move the network subsystem to the NIC
- Hardware acceleration of the dispatcher
- Automatic inference of workload characteristics
Thank you!

Questions?