Efficient Scheduling Policies for Microsecond-Scale Tasks

Sarah McClure  Amy Ousterhout  Sylvia Ratnasamy  Scott Shenker

UC Berkeley
sarah@eecs.berkeley.edu
### Datacenter Goals

- **Low Tail Latency**
- **High CPU Efficiency**

<table>
<thead>
<tr>
<th>App A</th>
<th>Low Latency?</th>
<th>High Efficiency?</th>
</tr>
</thead>
<tbody>
<tr>
<td>![CPU Image]</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>![CPU Image]</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Multiplex to Achieve Both Goals

**Goals:** low latency and high CPU efficiency

**Approach:** Quickly reallocate cores between applications
Multiplexing Systems
Lingering Inefficiency

- Memcached
- Batch App

- Total Achieved Throughput
- Offered Memcached Load
Lingering Inefficiency

![Diagram showing Memcached Throughput vs Total Normalized Throughput]

- **Memcached Throughput (Million Tasks/s)**
- **Total Normalized Throughput**
- **Ideal**

Legend:
- **Batch**
- **Memcached**
Lingering Inefficiency

![Chart showing Memcached Throughput and Total Normalized Throughput with Batch and Memcached categories. The chart illustrates the ideal scenario where the throughput is evenly distributed between the two categories.]
Lingering Inefficiency

Why do these systems fall short of perfect CPU efficiency?
Policy and Mechanism

Policies and implementation are coupled in existing systems
Challenge

Decouple system implementation details from policy choices to determine which policies perform best
Goal

Determine which load balancing and core allocation policies yield the best combination of latency and CPU efficiency for microsecond-scale tasks.
Our Approach

• Use simulations to determine the relative performance of policies without having to compare system implementations

• Model realistic overheads to move tasks and allocate cores

• Simplified model, but informative results

• Apply results to state-of-the-art systems
Key Findings

1. Work stealing performs best
2. Policies must proactively revoke cores for high efficiency
3. Beating the performance of static core allocations is hard
Key Findings

1. Work stealing performs best
2. Policies must proactively revoke cores for high efficiency
3. Beating the performance of static core allocations is hard

See the paper for all findings
System Model

- Tasks arrive from network or local CPU
- No preemption
- No a priori knowledge of task duration
Overheads

Load balancing
100 ns

Core allocation
5 us
Load Balancing Policies

- single queue (theoretical optimal)
- no load balancing
- enqueue choice
- work stealing
- work shedding
Which Load Balancing Policy is Best?

Give each policy the **same number of cycles** to work with

**Work stealing** achieves the best latency
Core Allocation Policies

- **theoretical optimum**
  - NP-hard
- **static allocation**
- **queueing delay**
- **utilization**
- **task arrivals**
- **failure to find work**
Core Allocation Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Signal</th>
</tr>
</thead>
</table>

20
Core Allocation Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caladan</td>
<td>Max queueing &gt; 5us, failure to find work</td>
</tr>
</tbody>
</table>
## Core Allocation Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caladan</td>
<td>Max queueing &gt; 5us, failure to find work</td>
</tr>
<tr>
<td>Per-Task</td>
<td>A task arrives, no available work</td>
</tr>
</tbody>
</table>
## Core Allocation Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caladan</td>
<td>Max queueing &gt; 5us, failure to find work</td>
</tr>
<tr>
<td>Per-Task</td>
<td>A task arrives, no available work</td>
</tr>
<tr>
<td><strong>Delay Range</strong></td>
<td>Average queueing delay</td>
</tr>
<tr>
<td><strong>Utilization Range</strong></td>
<td>Average CPU utilization</td>
</tr>
</tbody>
</table>
Load Balancing for Non-static Allocations?

Now, let’s **dynamically reallocate** cores

**Work stealing** still performs best
Which Allocation Policy is Best?

No single best policy

For simplicity, focus on work stealing

Policies which revoke cores proactively achieve better efficiency

Our policies are configurable and explicit
Experimental Results

Added Delay Range and Utilization Range to Caladan (uses work stealing)
Takeaways

Load balancing:
Work Stealing

Core Allocations:
Revoke proactively
Summary

• Used simulations with realistic overheads to evaluate policies
• Found that work stealing is the best load balancing policy
• Proposed two core allocation policies: delay range and utilization range
• Applied findings to Caladan to significantly improve CPU efficiency

Questions?
sarah@eecs.berkeley.edu

Code: https://github.com/smcclure20/scheduling-policies-sim
         https://github.com/shenango/caladan-policies