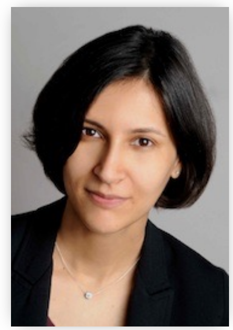
Efficient Scheduling Policies for Microsecond-Scale Tasks

Sarah McClure

Amy Ousterhout



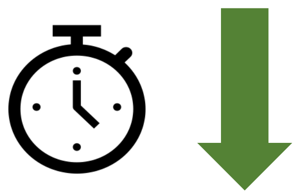
Sylvia Ratnasamy

Scott Shenker

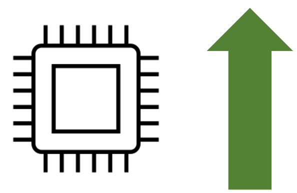
UC Berkeley

sarah@eecs.berkeley.edu

Datacenter Goals



Low Tail Latency

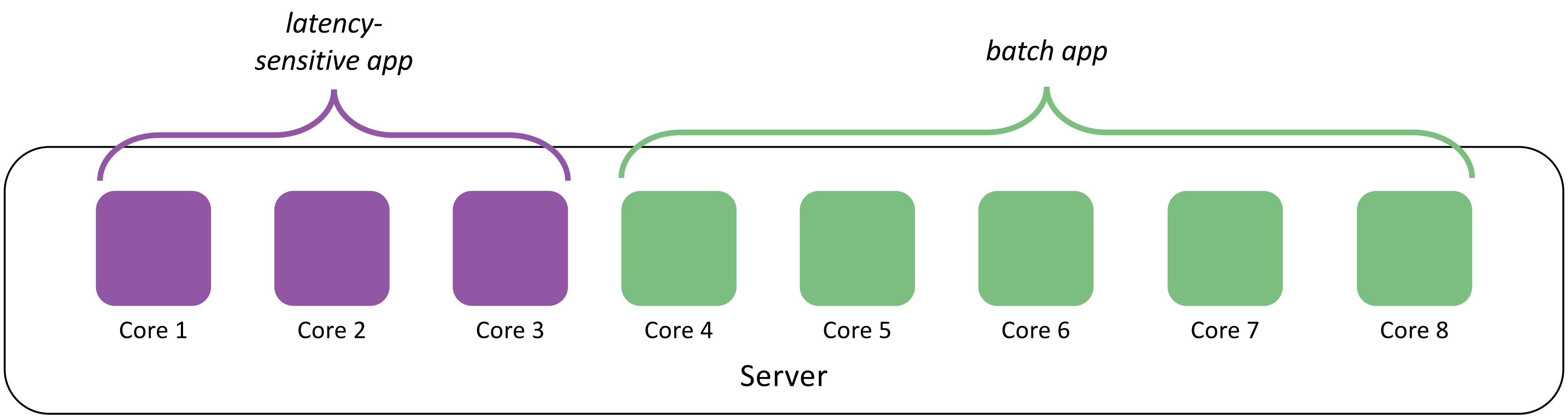


High CPU Efficiency

	Low Latency?	High Efficiency?
App A	✓	✗
App A	✗	✓

Multiplex to Achieve Both Goals

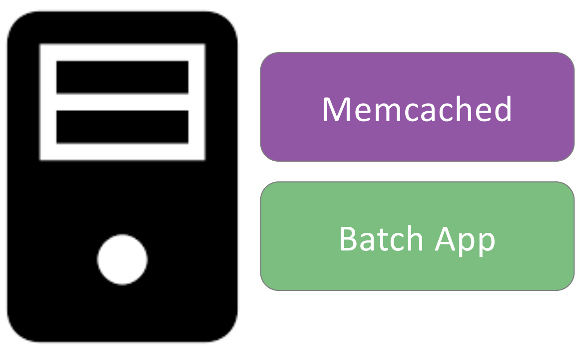
Goals: low latency and high CPU efficiency

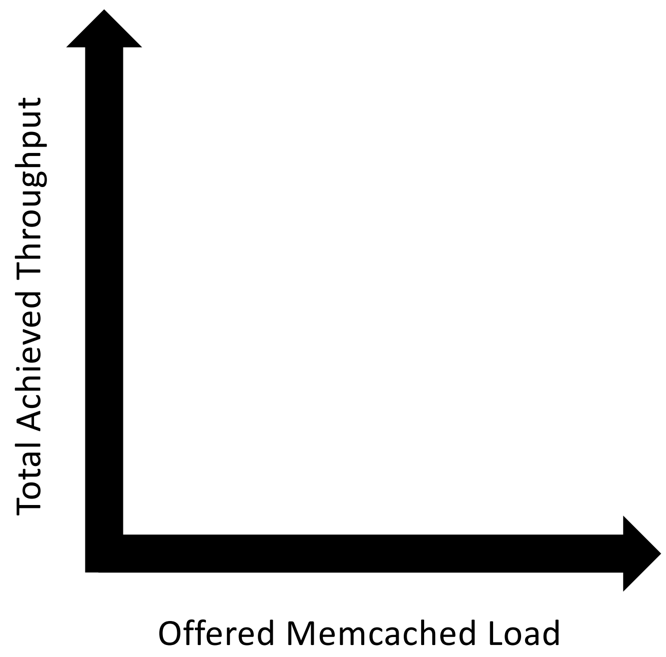


Approach: Quickly reallocate cores between applications

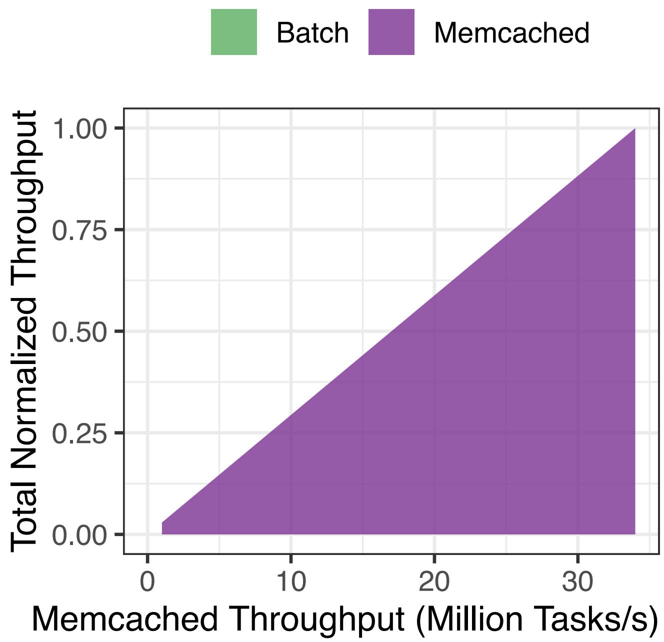
Multiplexing Systems

Lingering Inefficiency



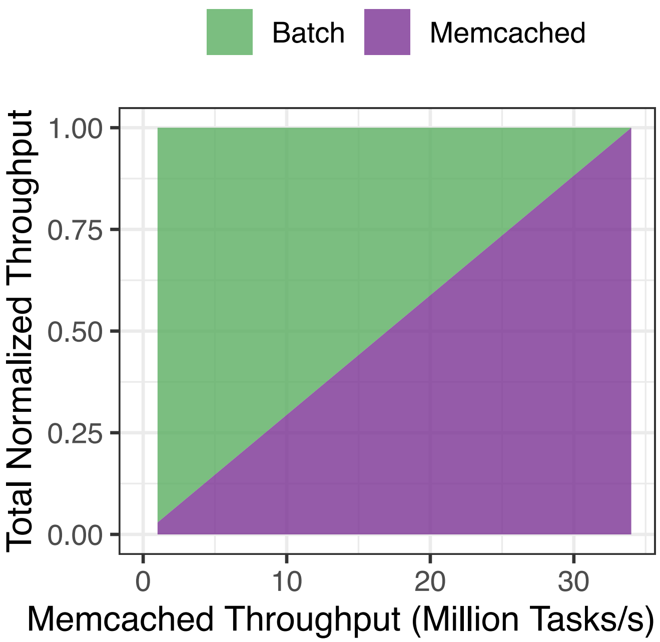


Lingering Inefficiency



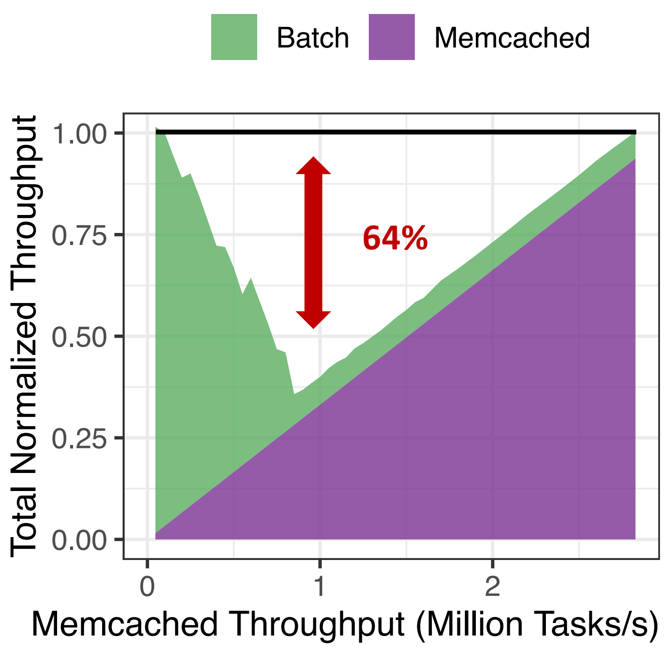
Ideal

Lingering Inefficiency

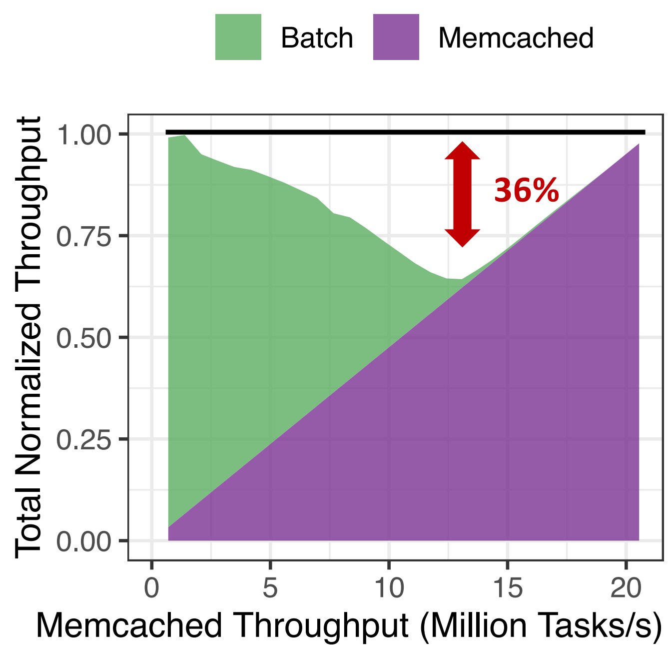


Ideal

Lingering Inefficiency



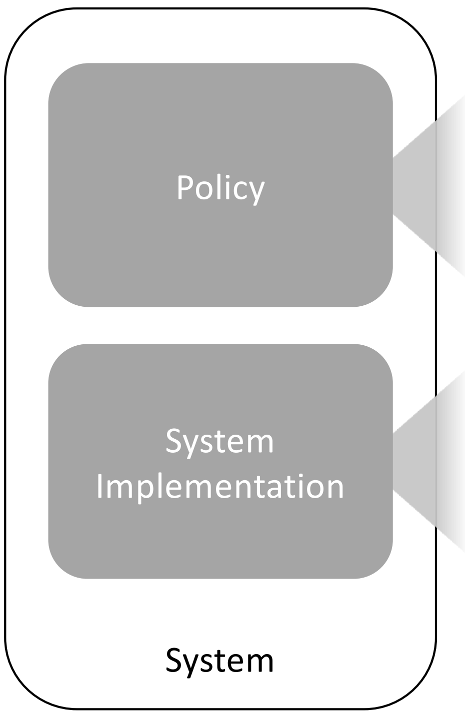
Arachne



Caladan

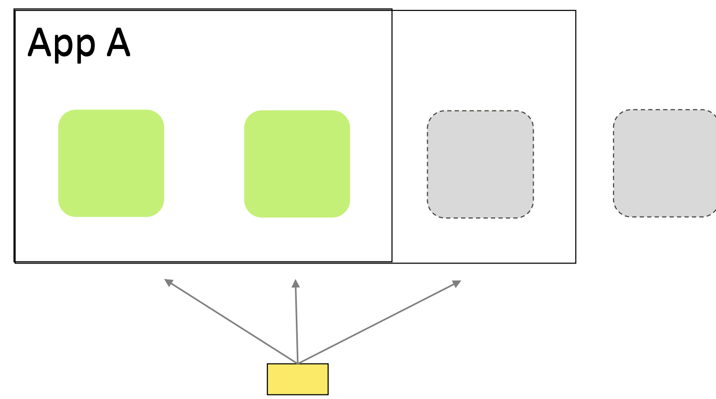
Why do these systems fall short of perfect CPU efficiency?

Policy and Mechanism



Core allocation?
Load balancing?

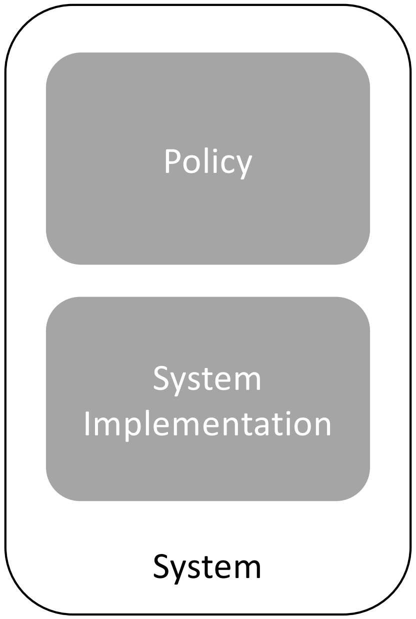
Kernel bypass?
Threading implementation?



Policies and implementation are coupled in existing systems

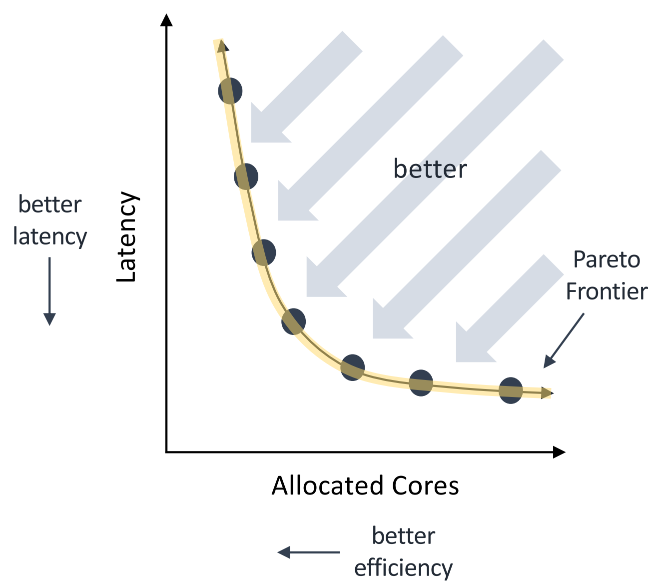
Challenge

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

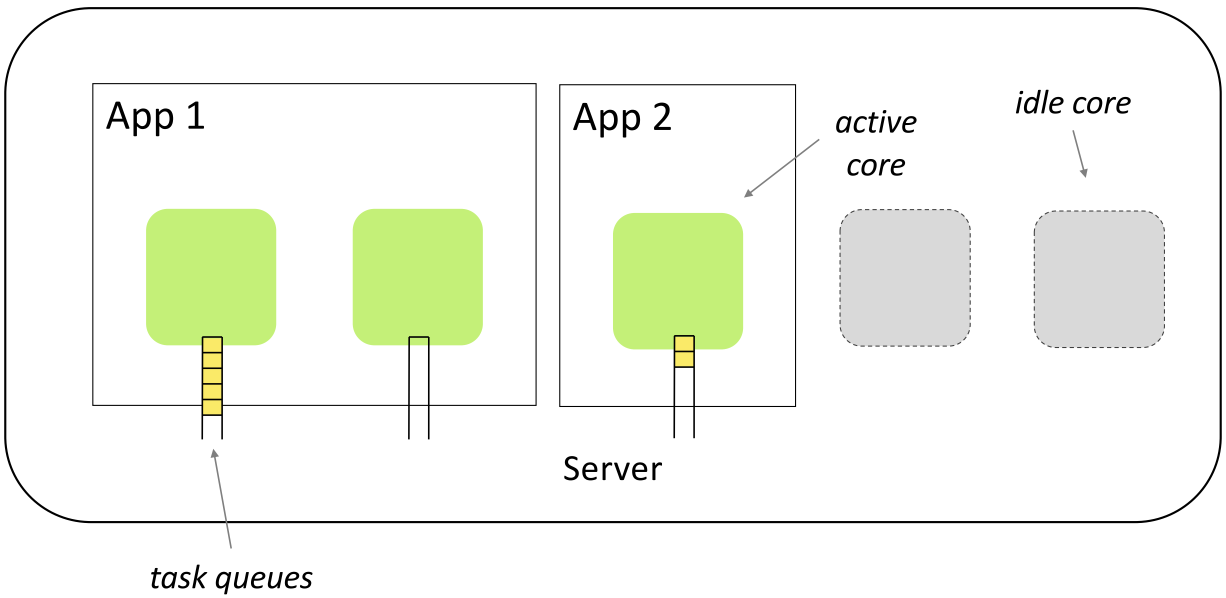
Key Findings

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

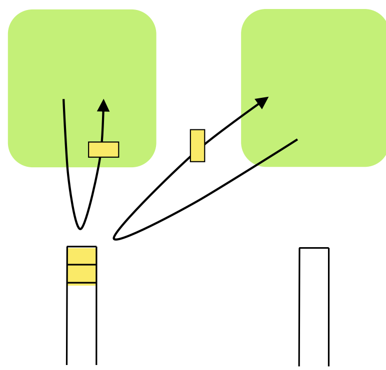
See the paper for all findings

System Model

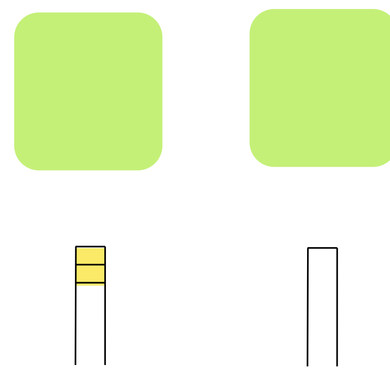
- Tasks arrive from network or local CPU
- No preemption
- No a priori knowledge of task duration



Overheads

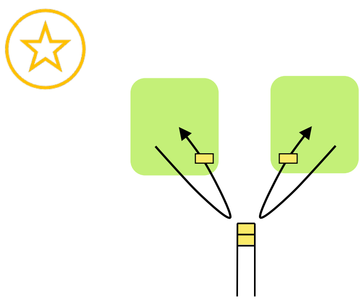


Load balancing
100 ns

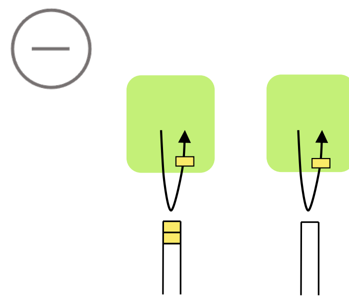


Core allocation
5 us

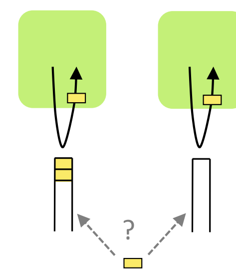
Load Balancing Policies



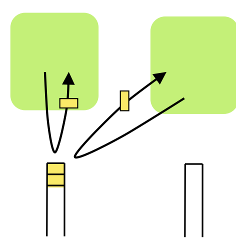
single queue
(theoretical optimal)



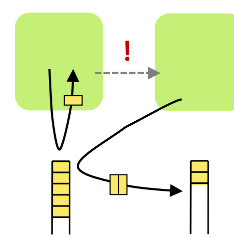
no load balancing



enqueue choice



work stealing



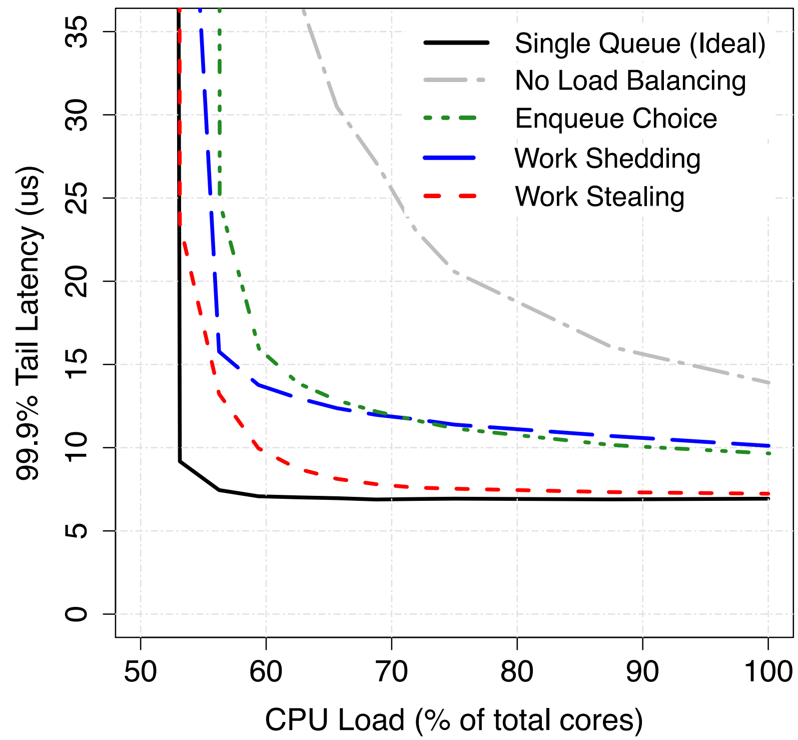
work shedding

Which Load Balancing Policy is Best?

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

Work stealing achieves the best latency

better latency



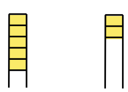
better efficiency

Core Allocation Policies

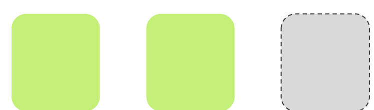
theoretical optimum
NP-hard

static allocation

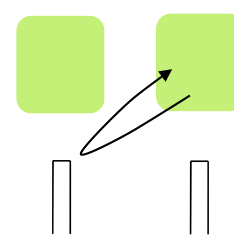
task arrivals



queueing delay



utilization



failure to find work

Core Allocation Policies

Policy	Signal

Core Allocation Policies

Policy	Signal
Caladan	Max queueing > 5us, failure to find work

Core Allocation Policies

Policy	Signal
Caladan	Max queueing > 5us, failure to find work
Per-Task	A task arrives, no available work

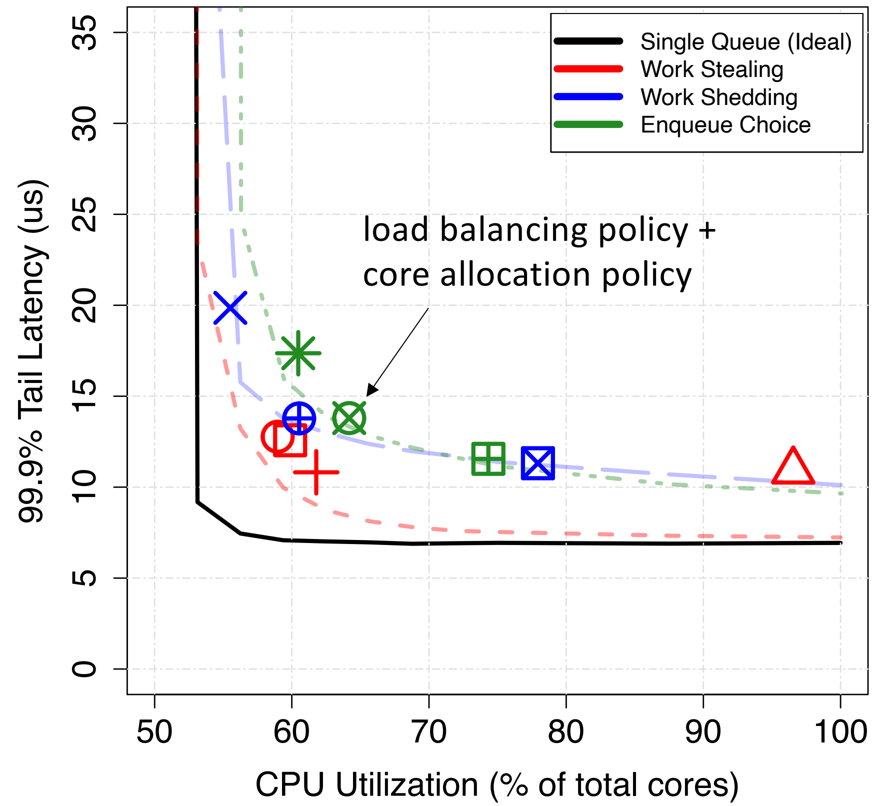
Core Allocation Policies

Policy	Signal
Caladan	Max queueing > 5us, failure to find work
Per-Task	A task arrives, no available work
Delay Range	Average queueing delay
Utilization Range	Average CPU utilization

Load Balancing for Non-static Allocations?

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

Work stealing still performs best



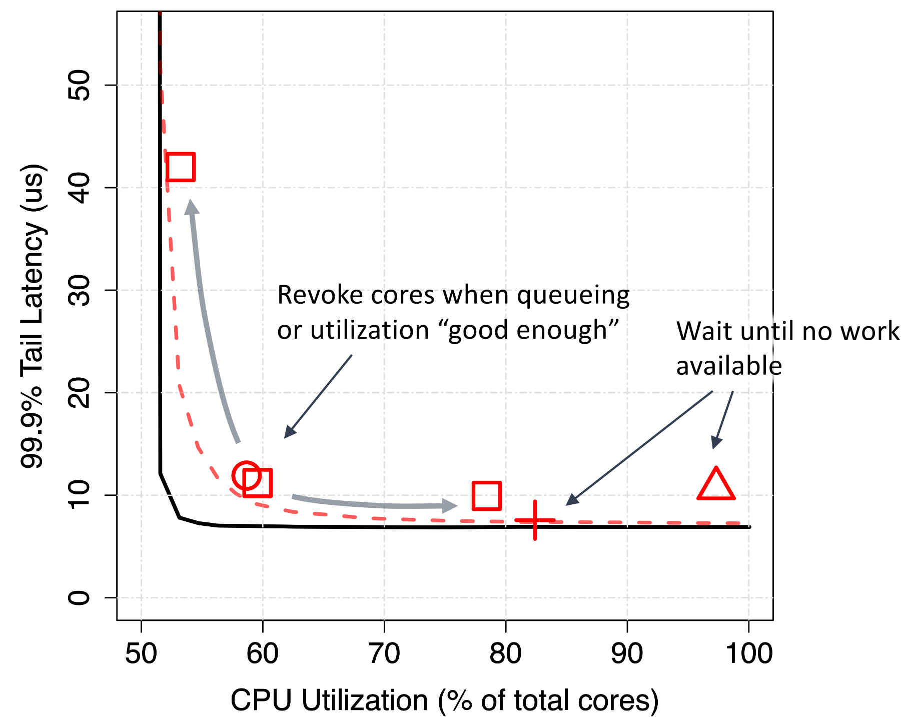
Which Allocation Policy is Best?

No single best policy

For simplicity, focus on work stealing

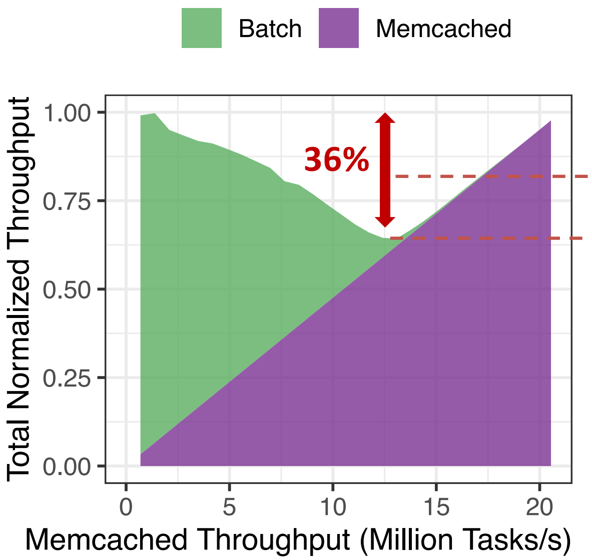
Policies which **revoke cores proactively** achieve better efficiency

Our policies are **configurable and explicit**

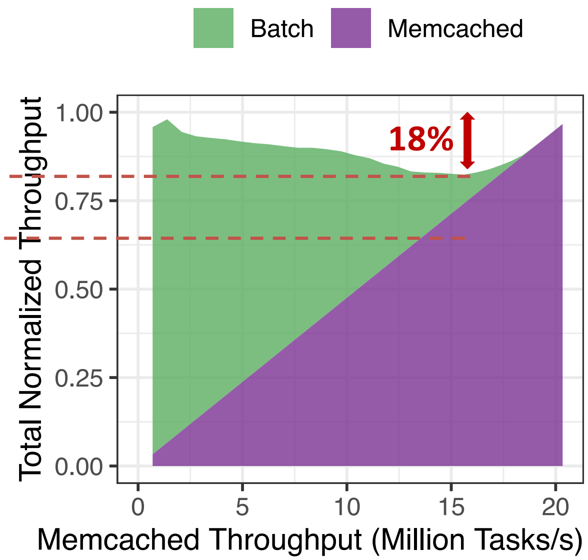


Experimental Results

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



Caladan



Delay Range

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