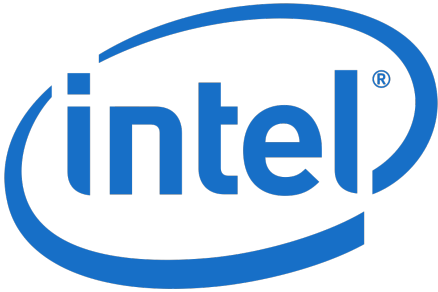


µTune: Auto-Tuned Threading for OLDI Microservices

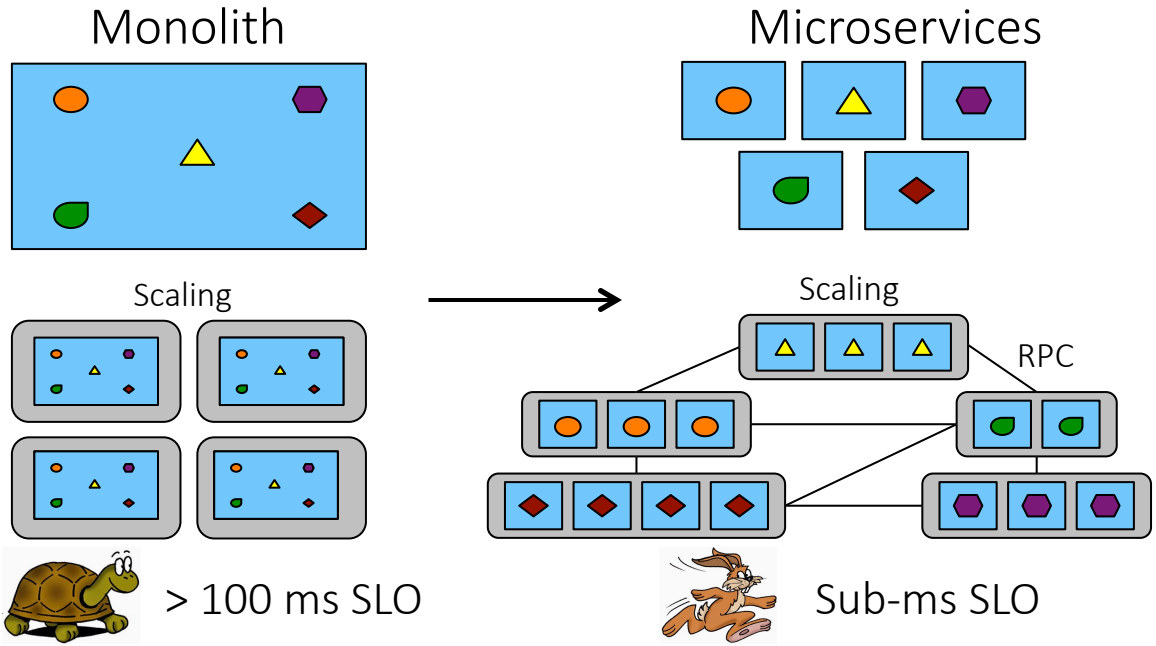
Akshitha Sriraman, Thomas F. Wenisch

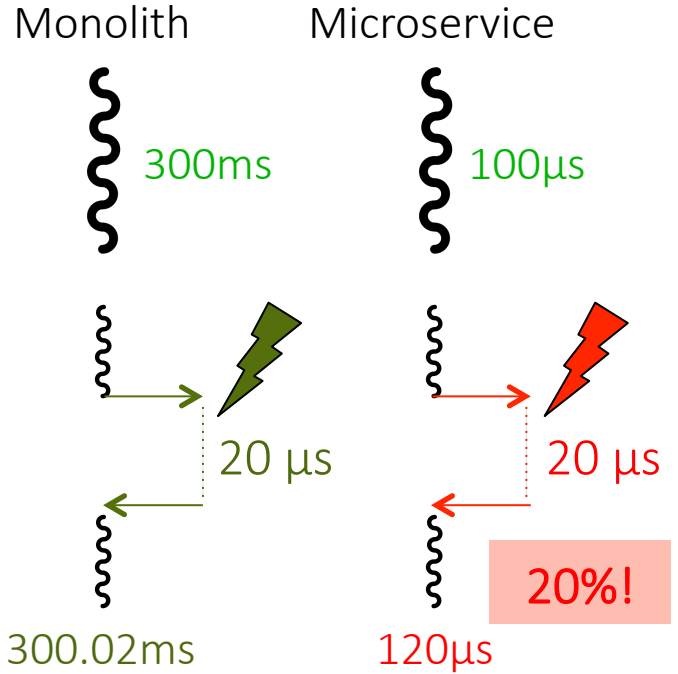
University of Michigan

OSDI 2018

10/08/2018

OLDI: From Monoliths to Microservices

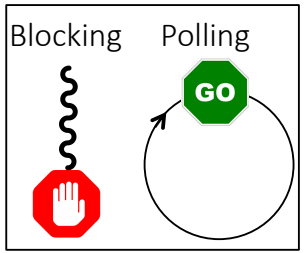




Sub-ms-scale system overheads must be characterized for microservices

Impact of Threading on Microservices

- Our focus: Sub-ms overheads due to threading design

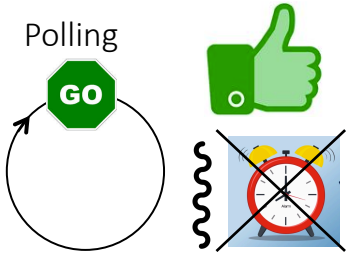


Lock contention

Thread wakeups

Spurious context switch

Low load →



High load →

Threading model that exhibits best tail latency is a function of load

Contributions

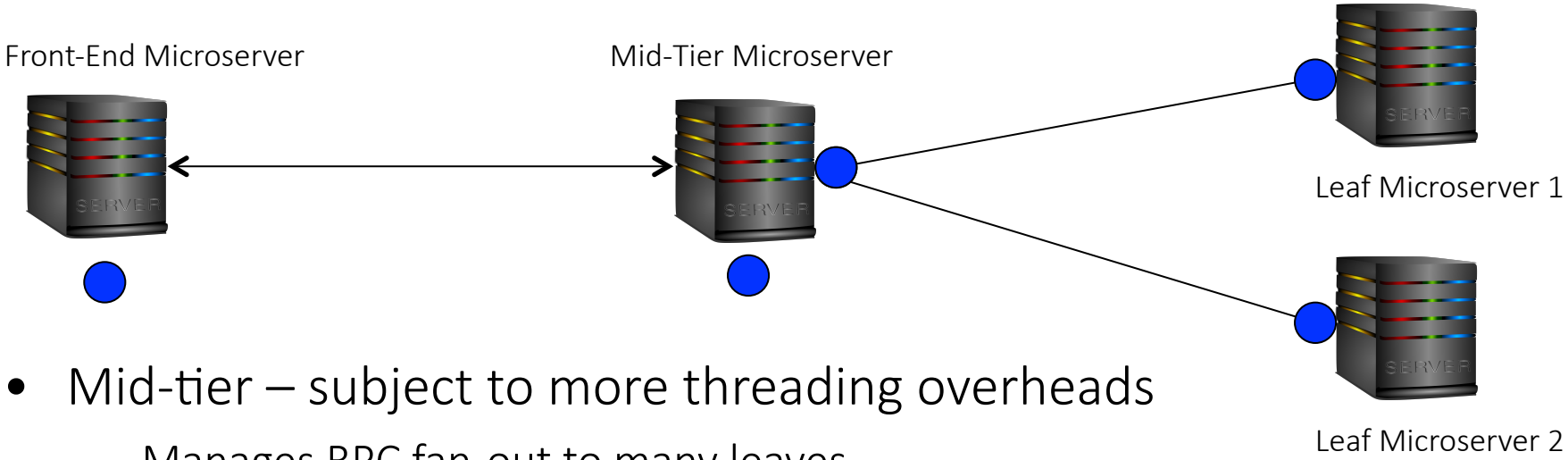
- A taxonomy of threading models
  - Structured understanding of threading implications
  - ➢ Reveals tail inflection points across load
  - ➢ Peak load-sustaining model is worse at low load
- µTune:
  - Uses tail inflection insights to optimize tail latency
  - Tunes model & thread pool size across load
  - Simple interface: Abstracts threading model from RPC code

Up to **1.9x** tail improvement over static throughput-optimized model

Outline

- Motivation
- A taxonomy of threading models
- µTune:
  - Simple interface design
  - Automatic load adaptation system
- Evaluation

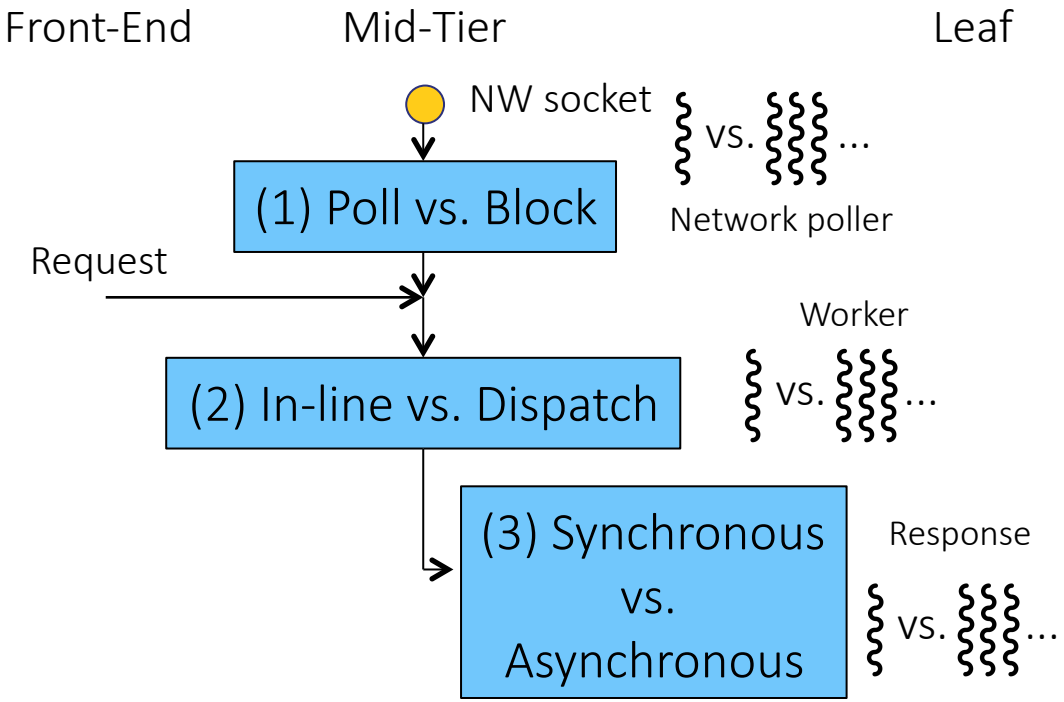
Mid-tier Faces More Threading Overheads



- Mid-tier – subject to more threading overheads
  - Manages RPC fan-out to many leaves
  - RPC layer interactions dominate computation

Threading overheads must be characterized for ***mid-tier*** microservices

A Taxonomy of Threading Models



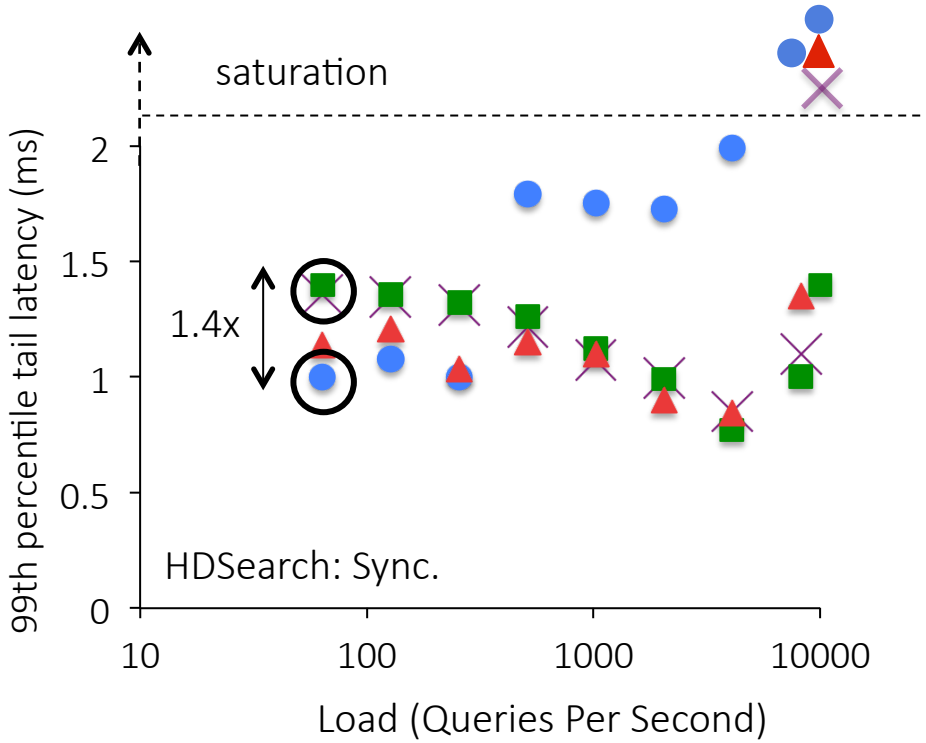
Synchronous

	Block	Poll
In-line	SIB	SIP
Dispatch	SDB	SDP

Asynchronous

	Block	Poll
In-line	AIB	AIP
Dispatch	ADB	ADP

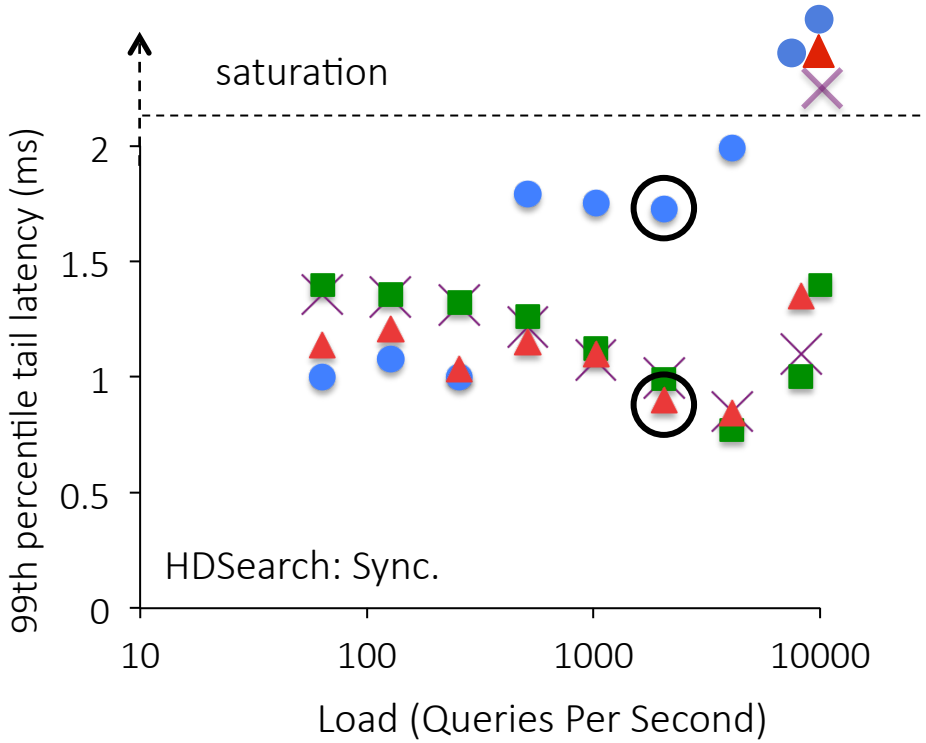
Latency Tradeoffs Across Threading Models



In-line Block
In-line Poll
Dispatch Block
Dispatch Poll

In-line Poll has lowest low-load latency: Avoids thread wakeup delays

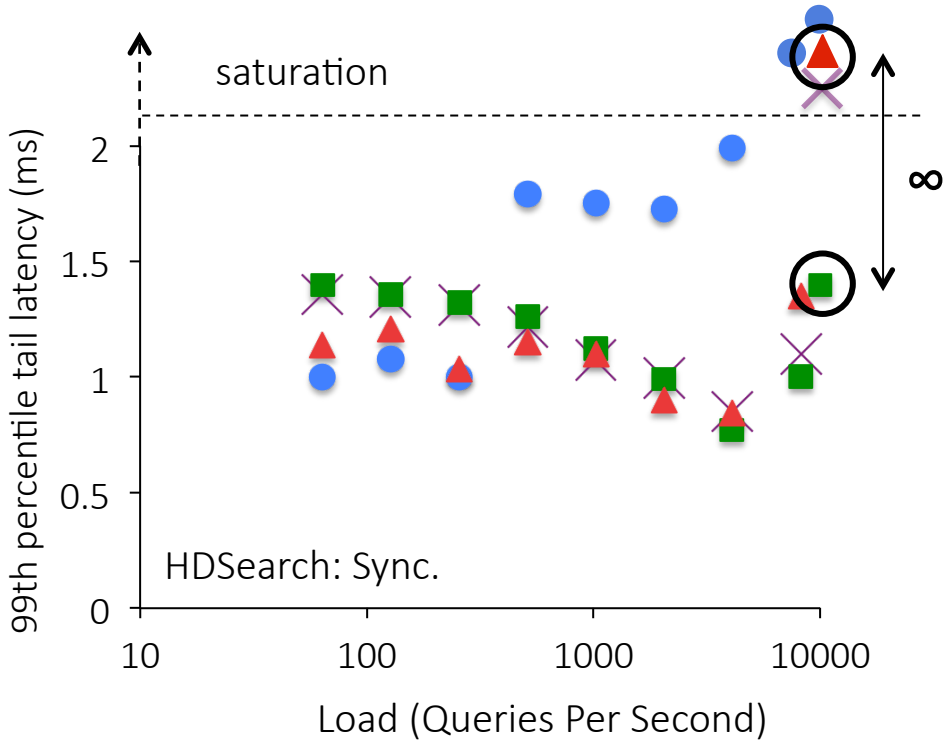
Latency Tradeoffs Across Threading Models

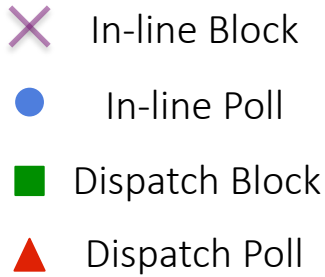


In-line Block
In-line Poll
Dispatch Block
Dispatch Poll

In-Line Poll faces contention; Dispatch Poll with one network poller is best

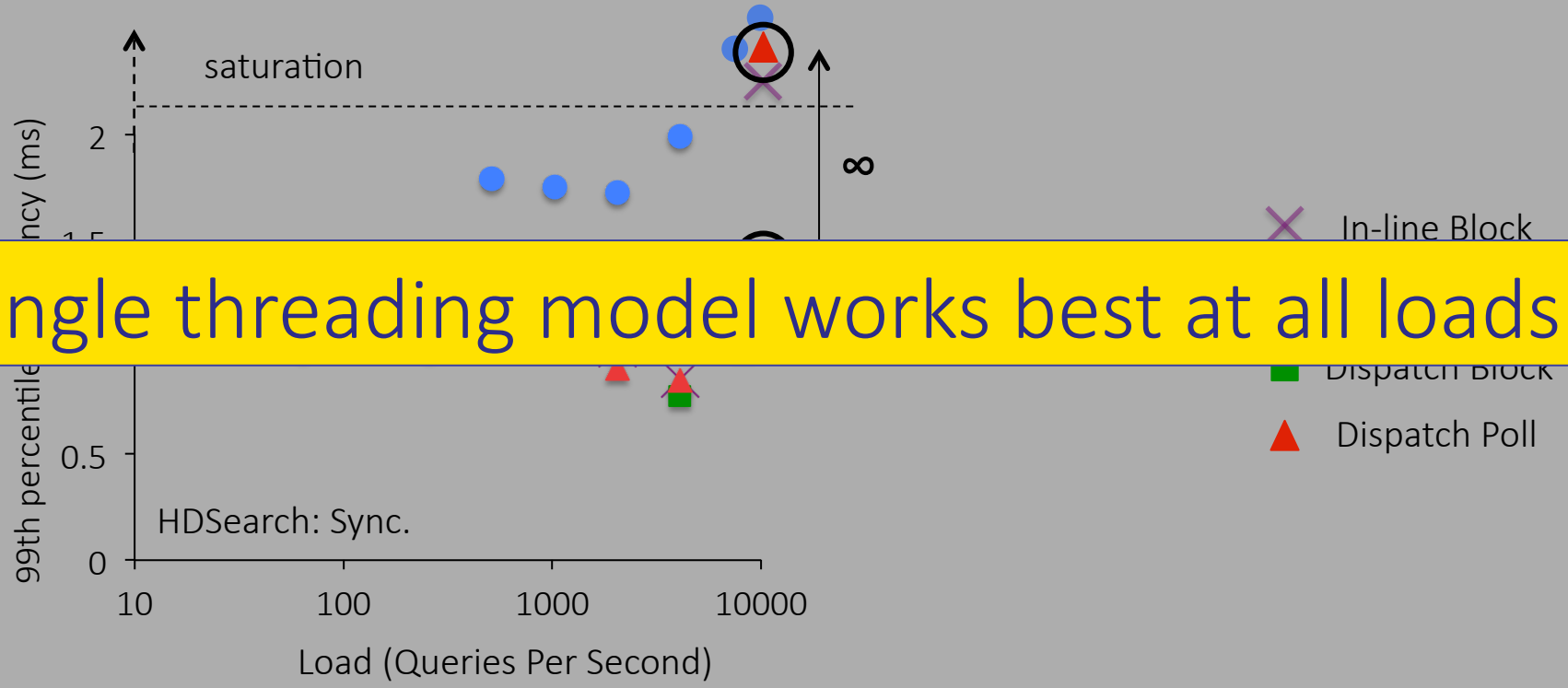
Latency Tradeoffs Across Threading Models





Dispatch Block is best at high load as it does not waste CPU

Latency Tradeoffs Across Threading Models



No single threading model works best at all loads

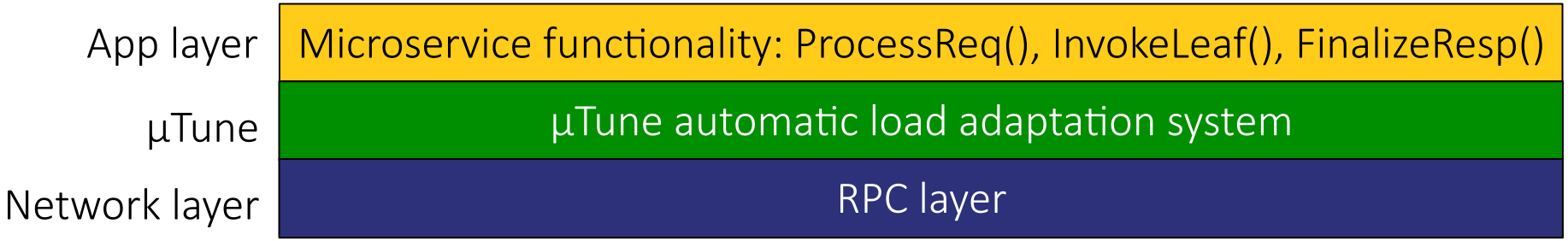
Need for Automatic Load Adaptation: µTune

- Threading choice can significantly affect tail latency
- Threading latency trade-offs are not obvious
- Most software face latency penalties due to static threading

Opportunity: Exploit trade-offs among threading models at run-time

µTune

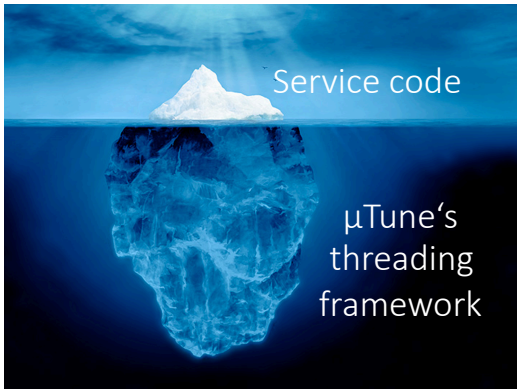
- Load adaptation: Vary threading model & pool size at run-time
- Abstract threading model boiler-plate code from RPC code



Simple interface: Developer defines only three functions

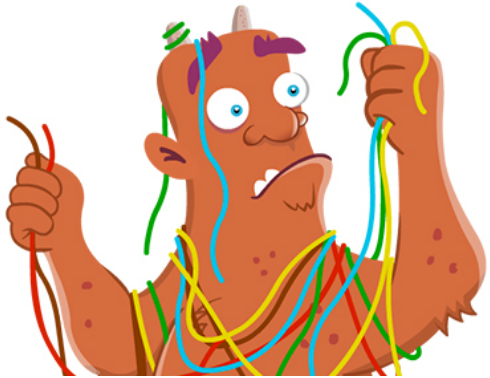
µTune: Goals & Challenges

Simple interface



Quick load change detection

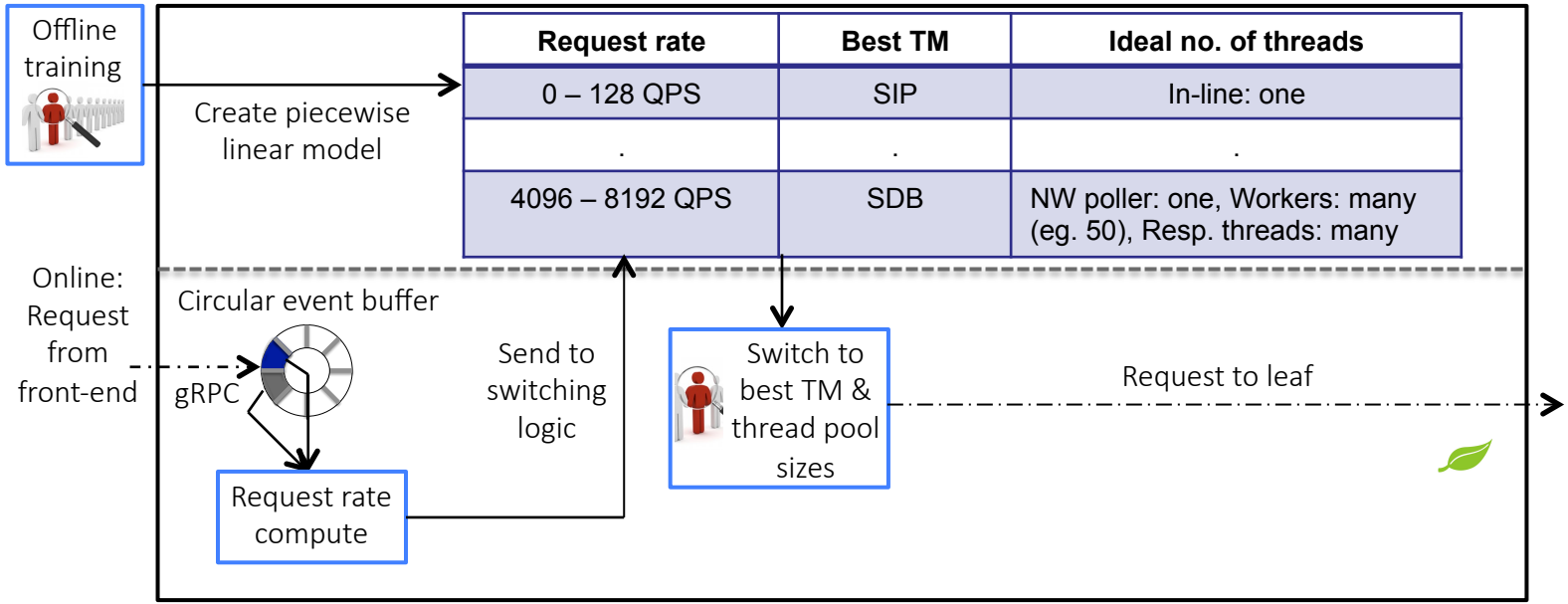
Fast threading model switches



Scale thread pools

µTune System Design: Auto-Tuner

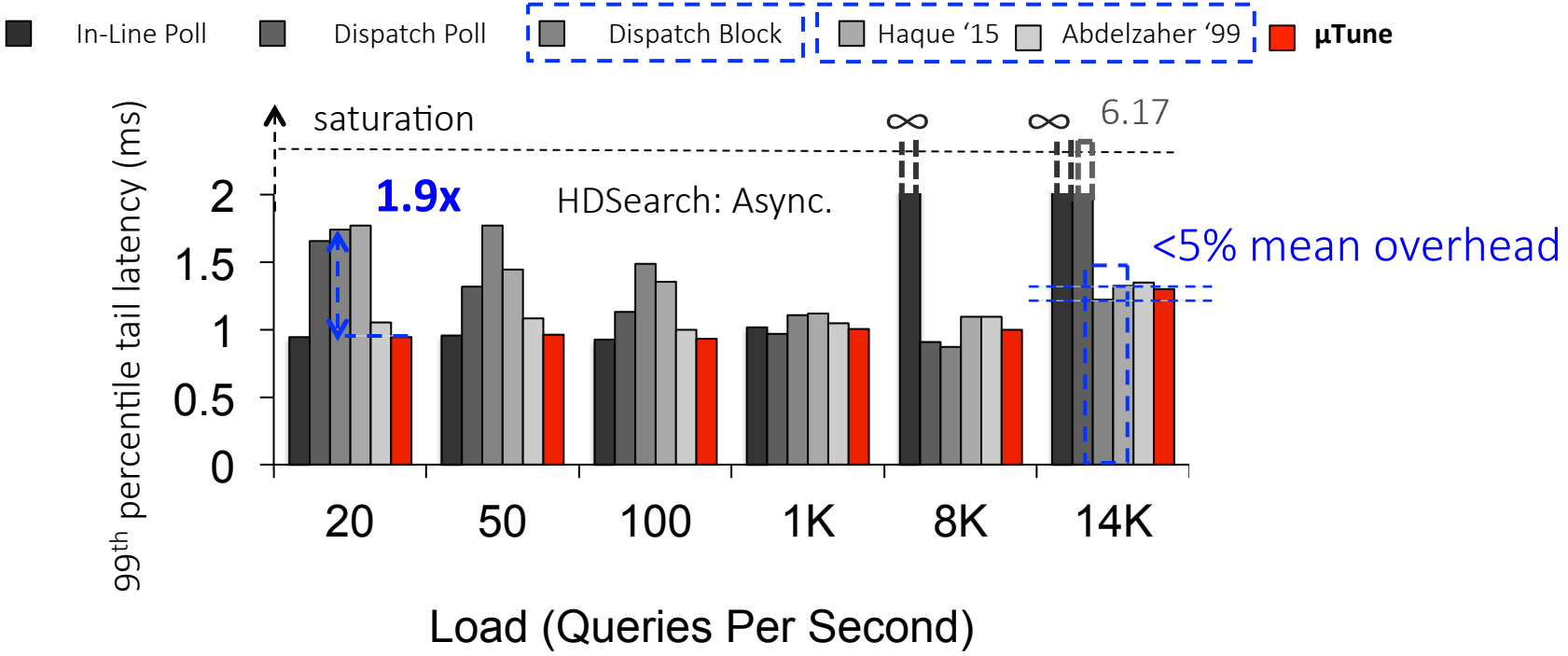
- Dynamically picks threading model & pool sizes based on load



Experimental Setup

- µSuite [Sriraman '18] benchmark suite:
  - Load generator, a mid-tier, 4 or 16 leaf microservers
- Study µTune's adaptation in two load scenarios:
  - Steady-state
  - Transient

Evaluation: µTune's Load Adaptation



Converges to best threading model & pool sizes to improve tails by up to **1.9x**

Conclusion

- Taxonomy of threading models
  - No single threading model has best tail latency across all load
- µTune – threading model framework + load adaptation system
- Achieved up to **1.9x** tail speedup over best static model

µTune: Auto-Tuned Threading for OLDI Microservices

Akshitha Sriraman, Thomas F. Wenisch

University of Michigan

https://github.com/wenischlab/MicroTune

Poster number: 29