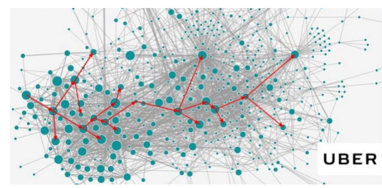
µManycore: A Cloud-Native CPU for Tail at Scale

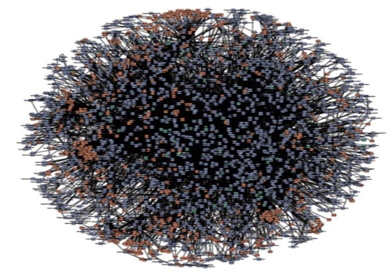
ISCA 2023

Jovan Stojkovic, Chunao Liu*, Muhammad Shahbaz*, Josep Torrellas
University of Illinois at Urbana-Champaign, *Purdue University

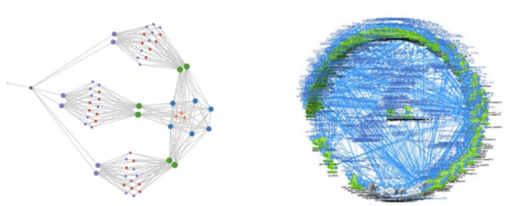
Emerging Software in the Cloud: Microservices

- Large monolithic applications decomposed into many small interdependent services
- Each service implements separate functionality
- Many benefits:
  - Scalability
  - Design simplicity
  - HW management



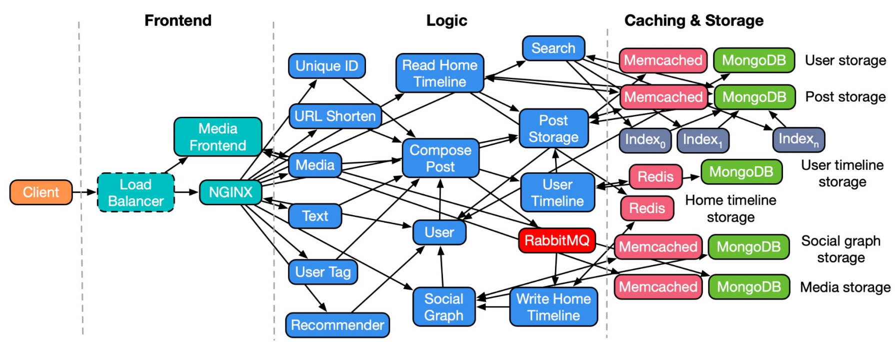


Structure of microservices at Amazon. Looks almost like a Death Star but is way more powerful.



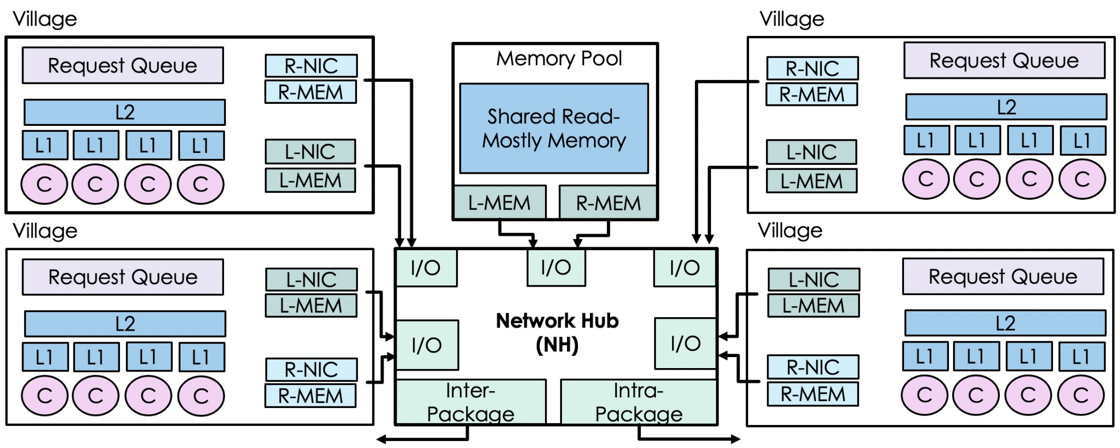
Simplified Architecture   Actual Architecture

Netflix architecture: simplified and actual scheme (source)



Contributions

- Characterization of microservice systems with conventional processors
- Propose **µManycore** – a processor architecture highly optimized for microservice workloads
  - Chiplet-based design with multiple small hardware cache-coherent domains
  - Hierarchical leaf-spine interconnection network on package
  - In-hardware request scheduling and context switching
- Tail latency reduction 10.4X, throughput improvement 15.5X

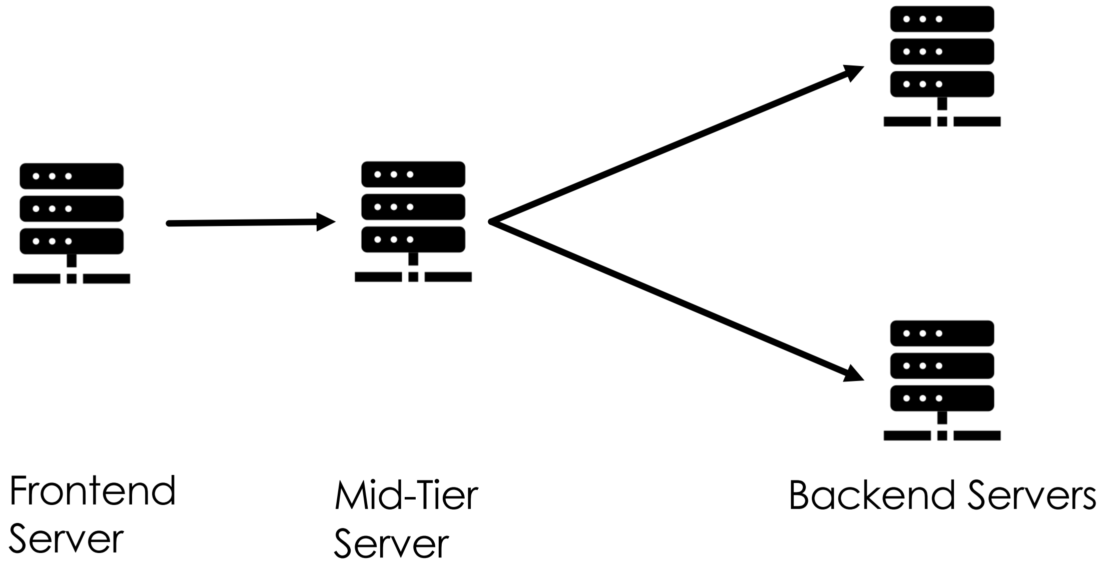


Mismatch Current Processors vs Microservices

Current Processors	Microservice Environments
Maximize average performance	Stringent tail latency constraints
Beefy processors	Many requests in parallel. Low instruction-level parallelism
Monolithic cache coherence	Microservices rarely share writable data
Optimized for long-running, predictable apps (prefetchers, branch predictors)	Short-running services; dynamic environment

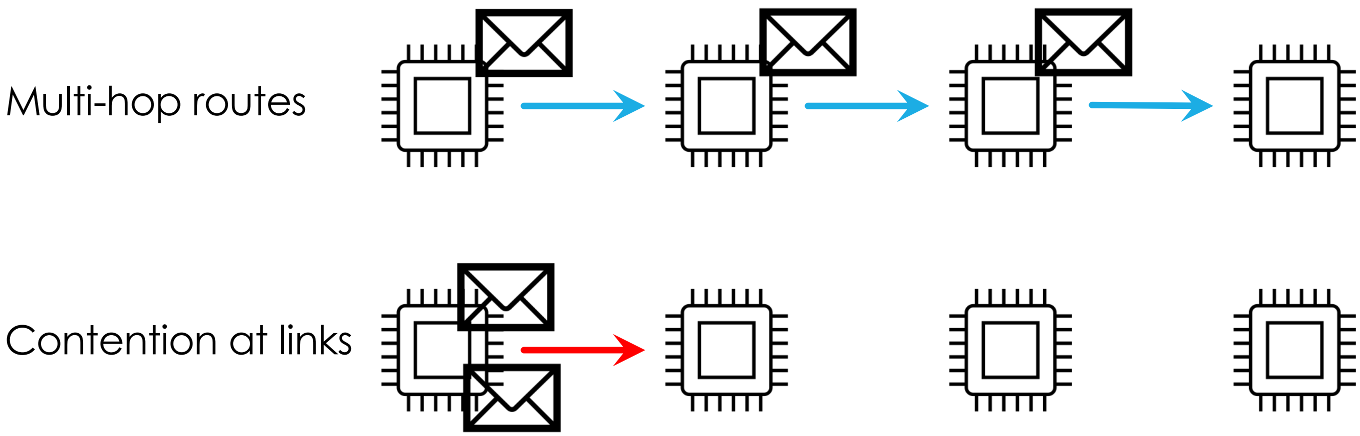
Designing Processors for Tail Latency

- Response time determined by the slowest service
- Identify and optimize away sources of contention
  - On-package network
  - Request queuing and scheduling
  - Context switching



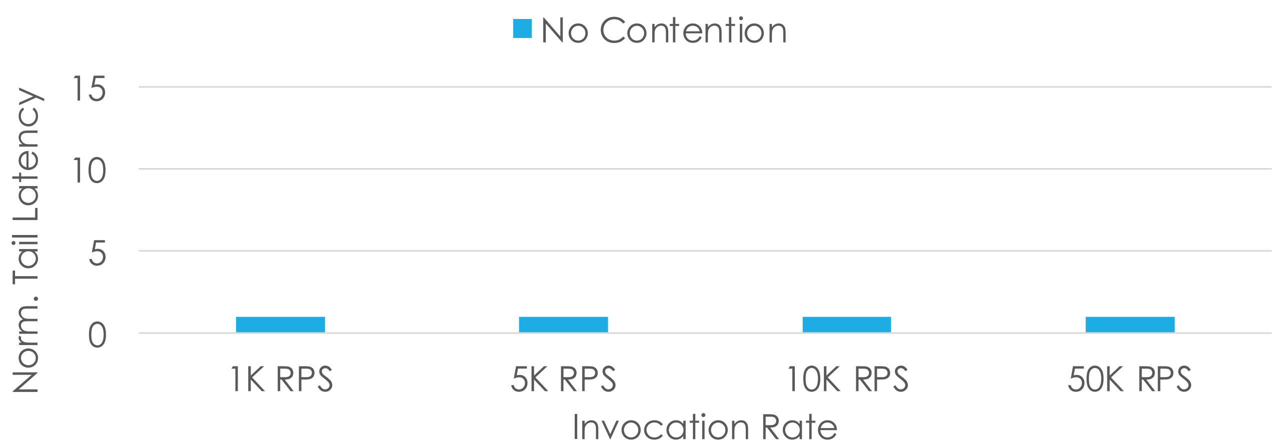
Hotspots in on-package network

- Inter-process communication due to RPCs and storage accesses
  - Lots of on-package messages



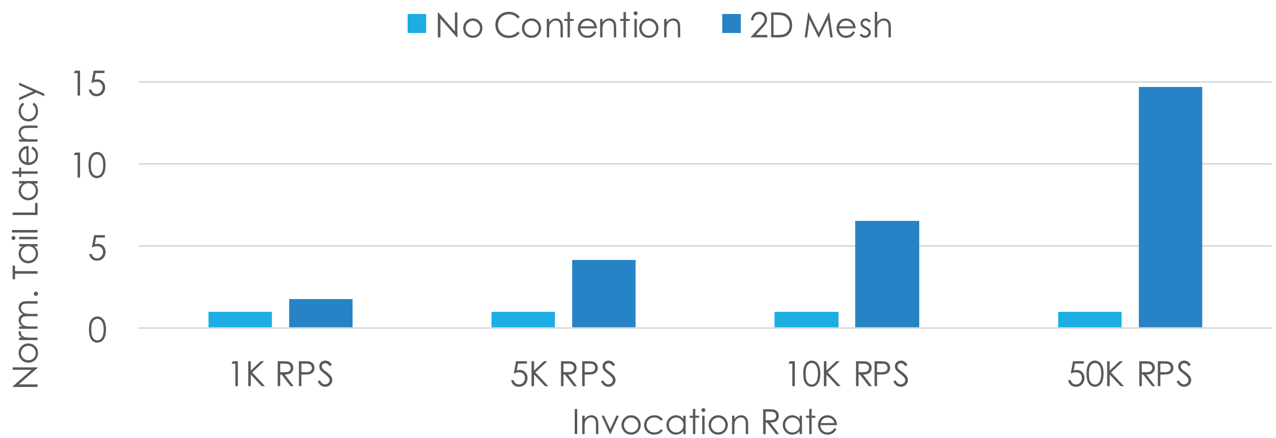
Hotspots in on-package network

- Inter-process communication due to RPCs and storage accesses
  - Lots of on-package messages
- **Contention at the on-package network can hurt the tail latency**



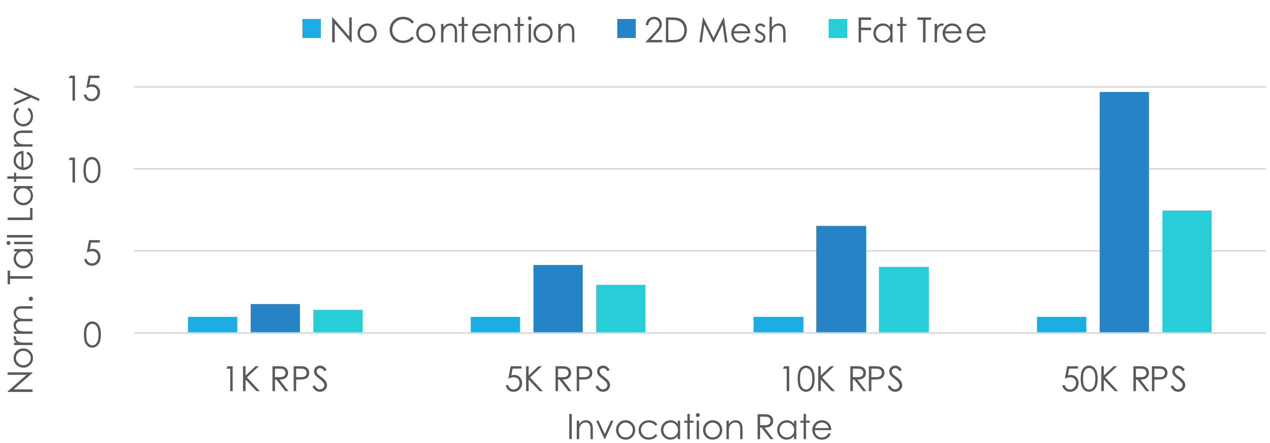
Hotspots in on-package network

- Inter-process communication due to RPCs and storage accesses
  - Lots of on-package messages
- **Contention at the on-package network can hurt the tail latency**



Hotspots in on-package network

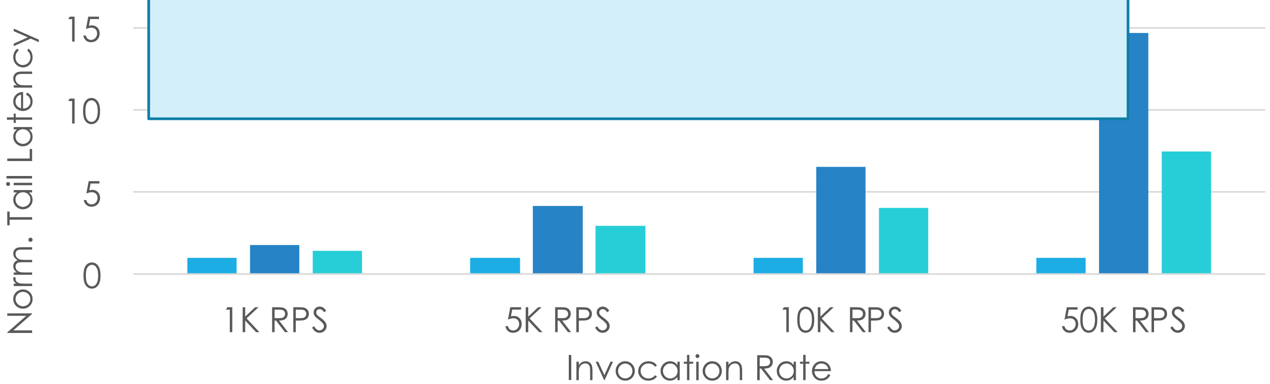
- Inter-process communication due to RPCs and storage accesses
  - Lots of on-package messages
- **Contention at the on-package network can hurt the tail latency**



Hotspots in on-package network

- Inter-process communication due to RPCs and storage accesses
  - Lots of on-pack
- **Contention at the**

We need a high-bandwidth and low-latency on-package network

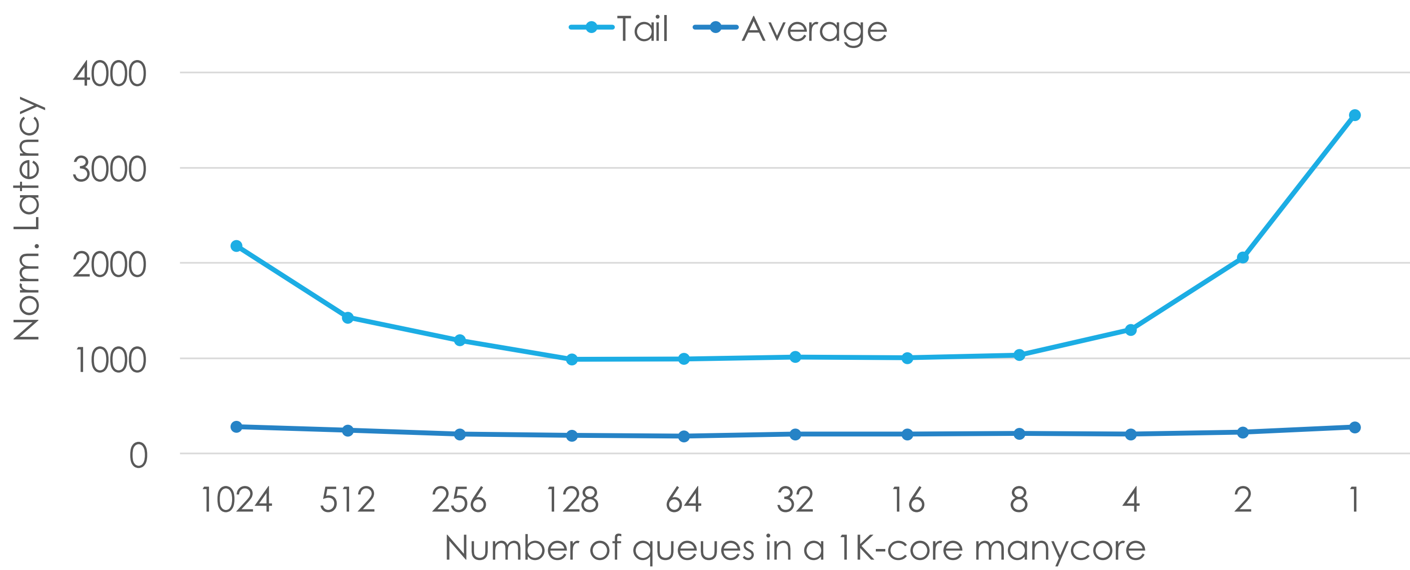


Hotspots in request queuing and scheduling

- Service requests come in bursts and need to be queued before execution
- **Design of the queueing system can impact tail latency**

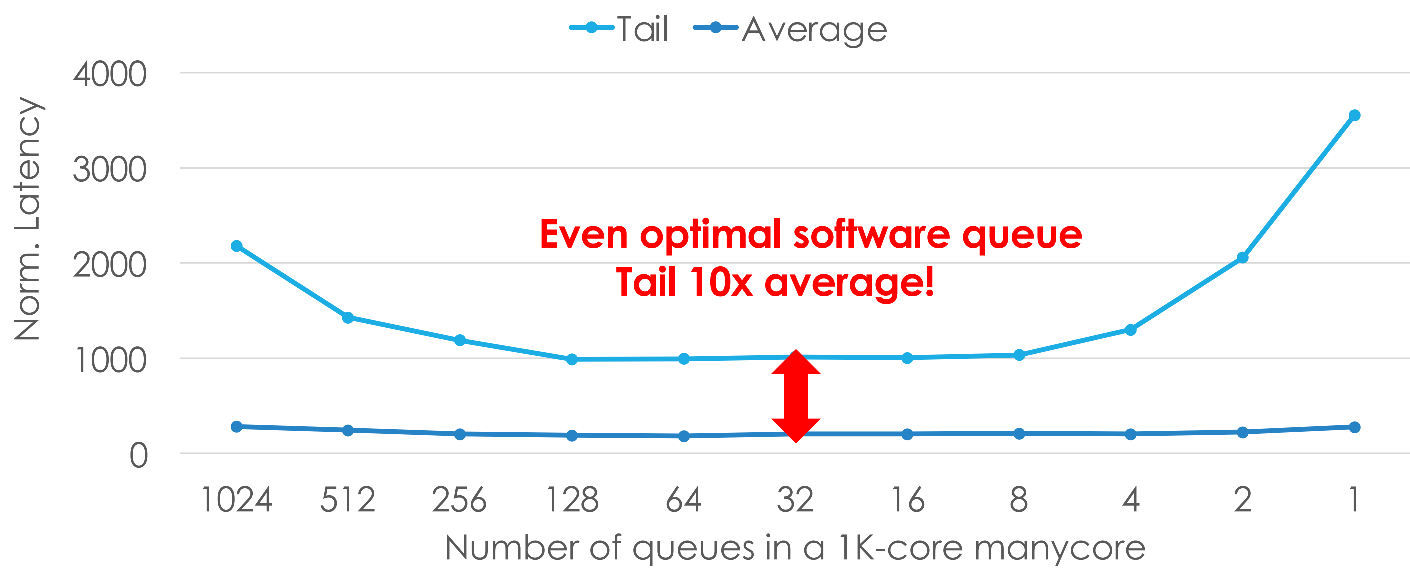
Hotspots in request queuing and scheduling

- Service requests come in bursts and need to be queued before execution
- **Design of the queueing system can impact tail latency**



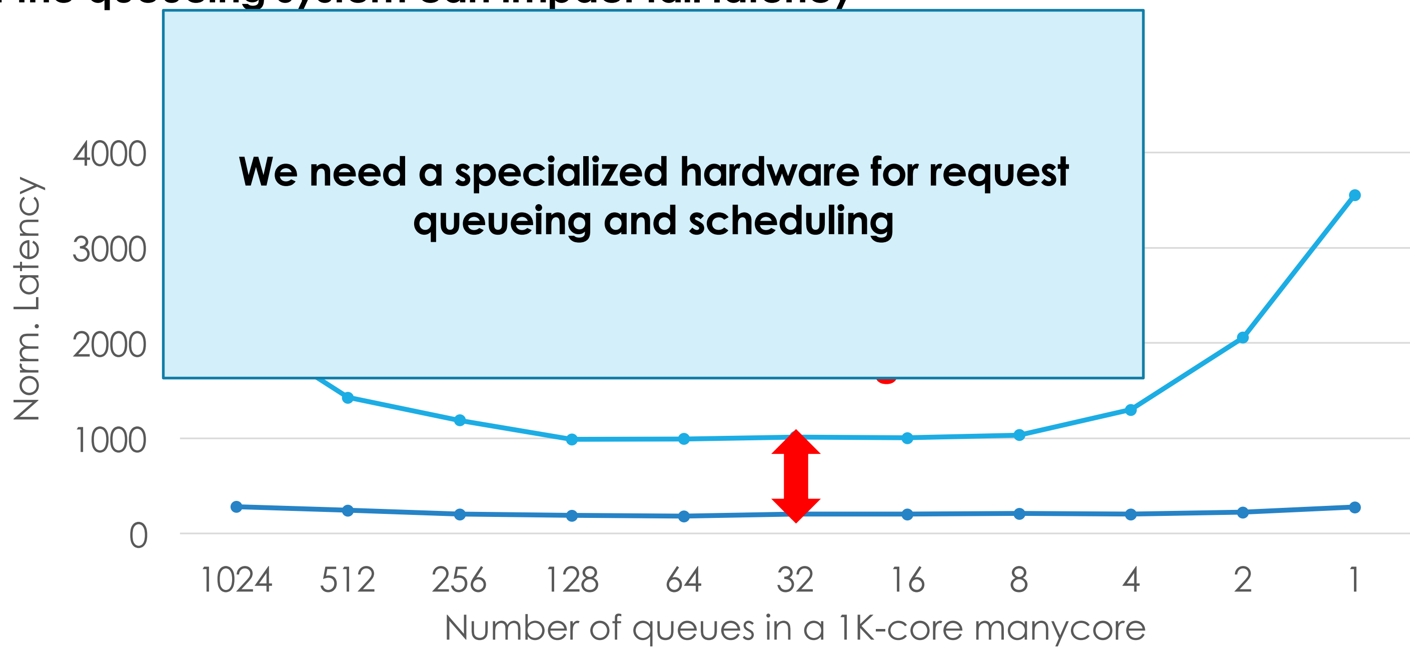
Hotspots in request queuing and scheduling

- Service requests come in bursts and need to be queued before execution
- **Design of the queueing system can impact tail latency**



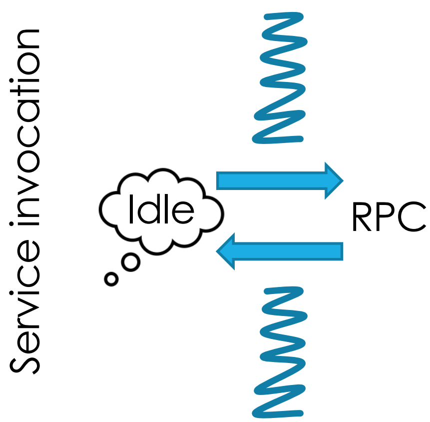
Hotspots in request queuing and scheduling

- Service requests come in bursts and need to be queued before execution
- **Design of the queueing system can impact tail latency**

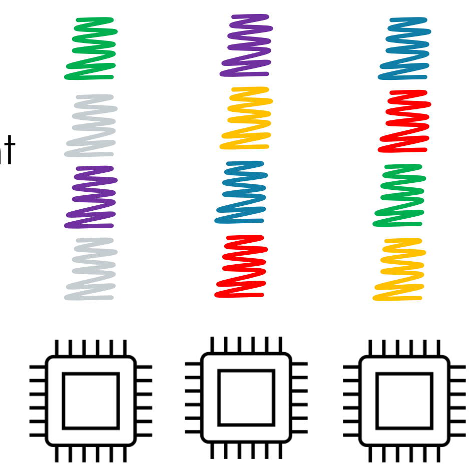


Hotspots in context switching

- Services spend majority of their execution time blocked, waiting on I/O
  - Remote storage accesses, or synchronous calls to other services

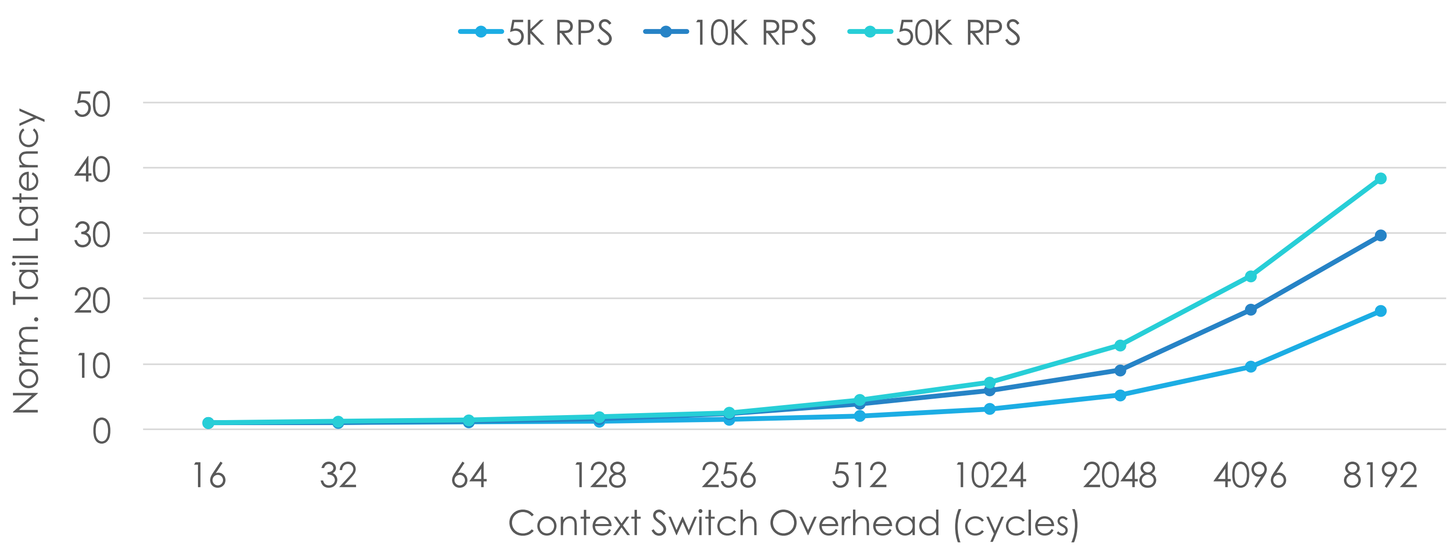


Need to perform frequent context switches!



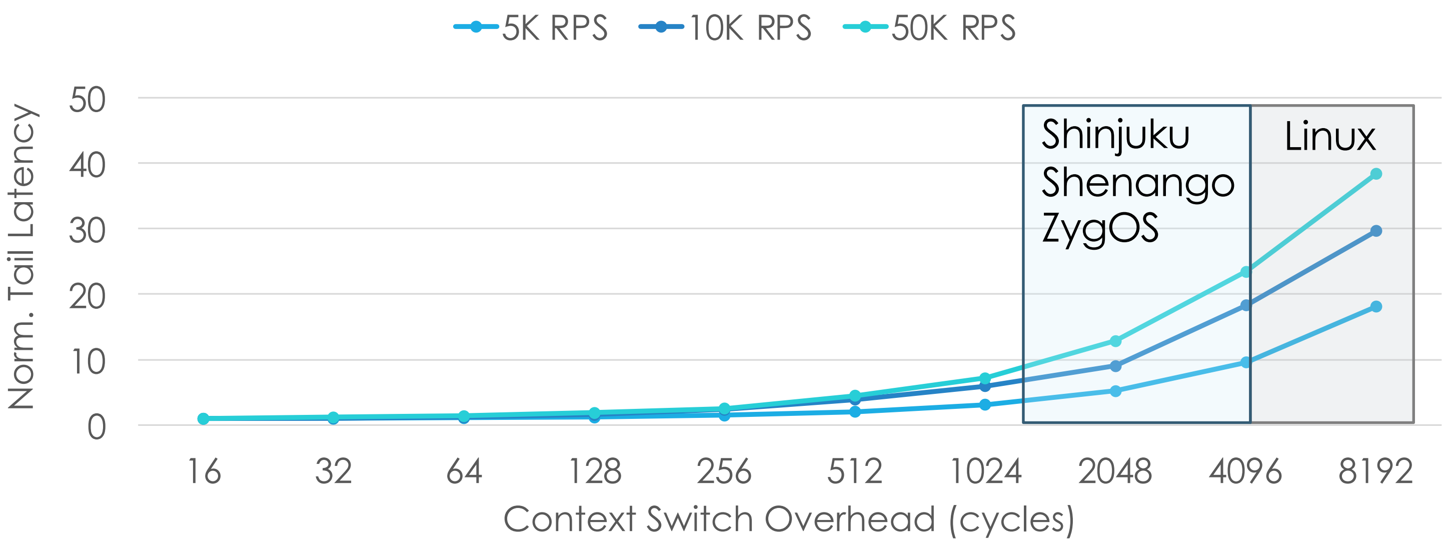
Hotspots in context switching

- Services spend majority of their execution time blocked, waiting on I/O
  - Remote storage accesses, or synchronous calls to other services



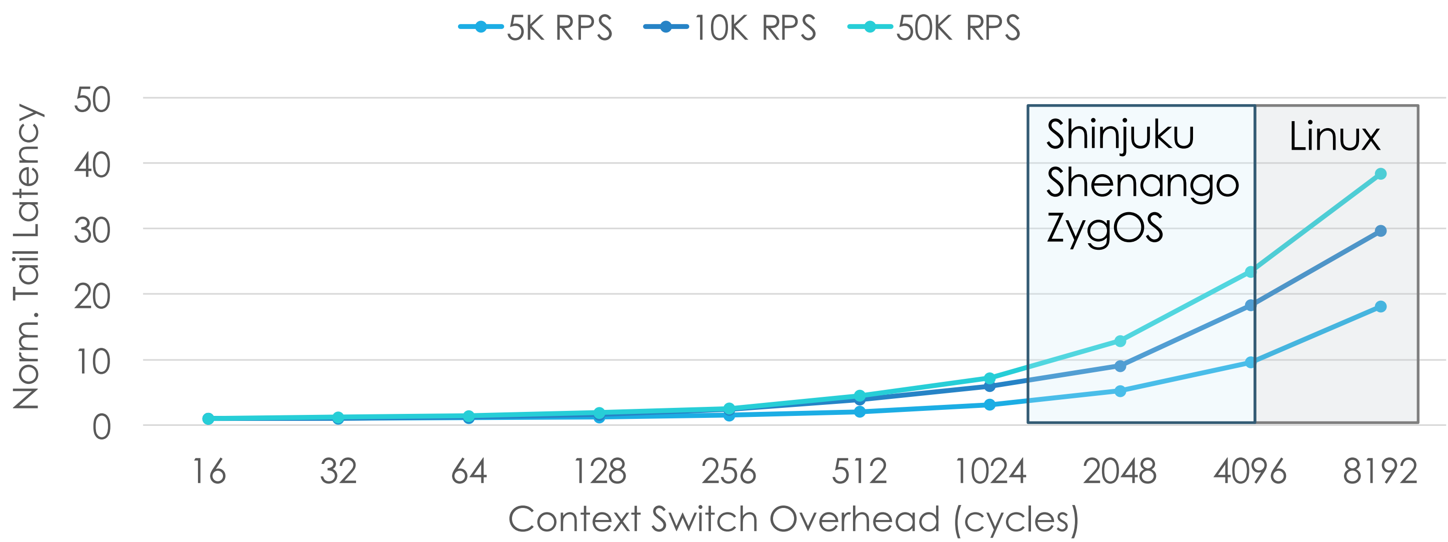
Hotspots in context switching

- Services spend majority of their execution time blocked, waiting on I/O
  - Remote storage accesses, or synchronous calls to other services



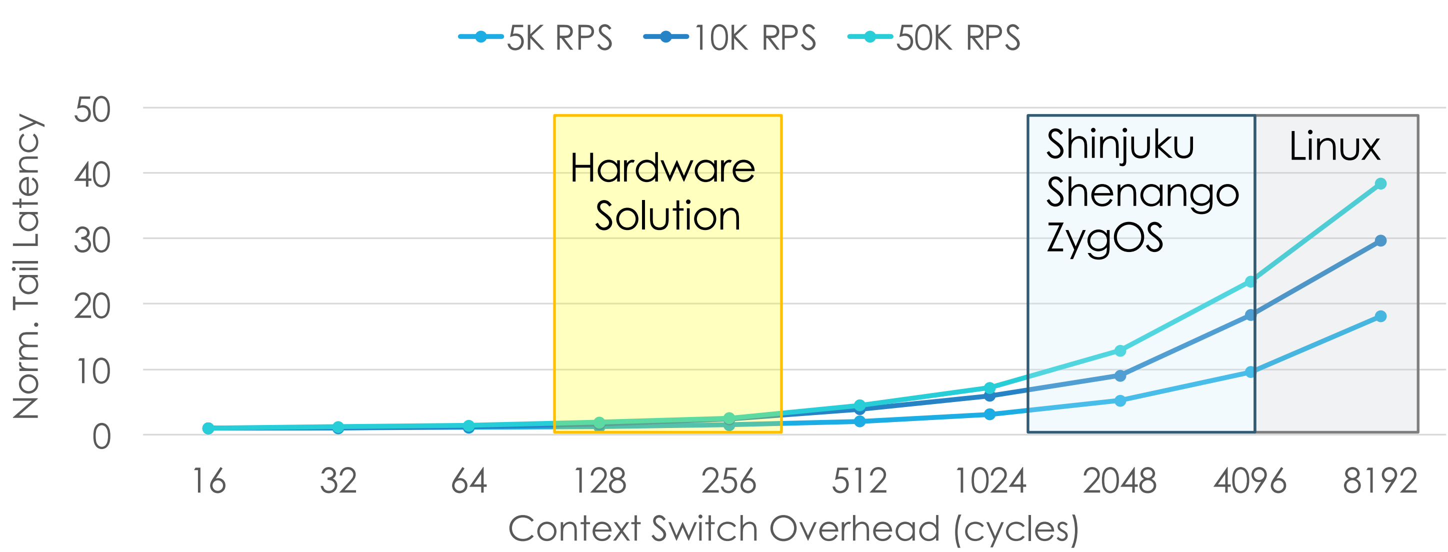
Hotspots in context switching

- **Even highly specialized software context switching penalty not negligible**



Hotspots in context switching

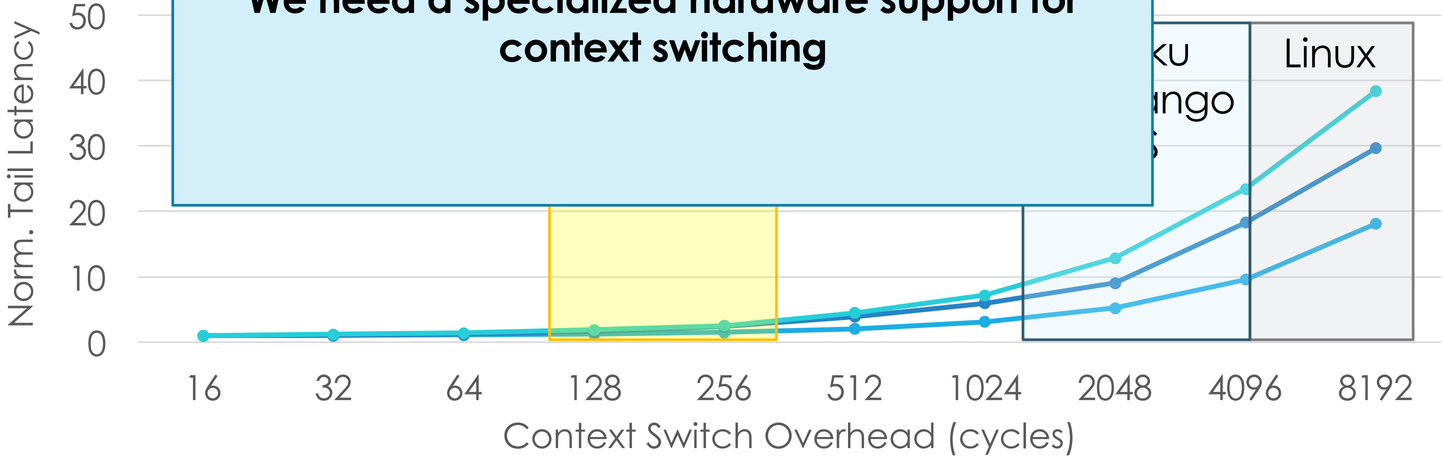
- Even highly specialized software context switching penalty not negligible



Hotspots in context switching

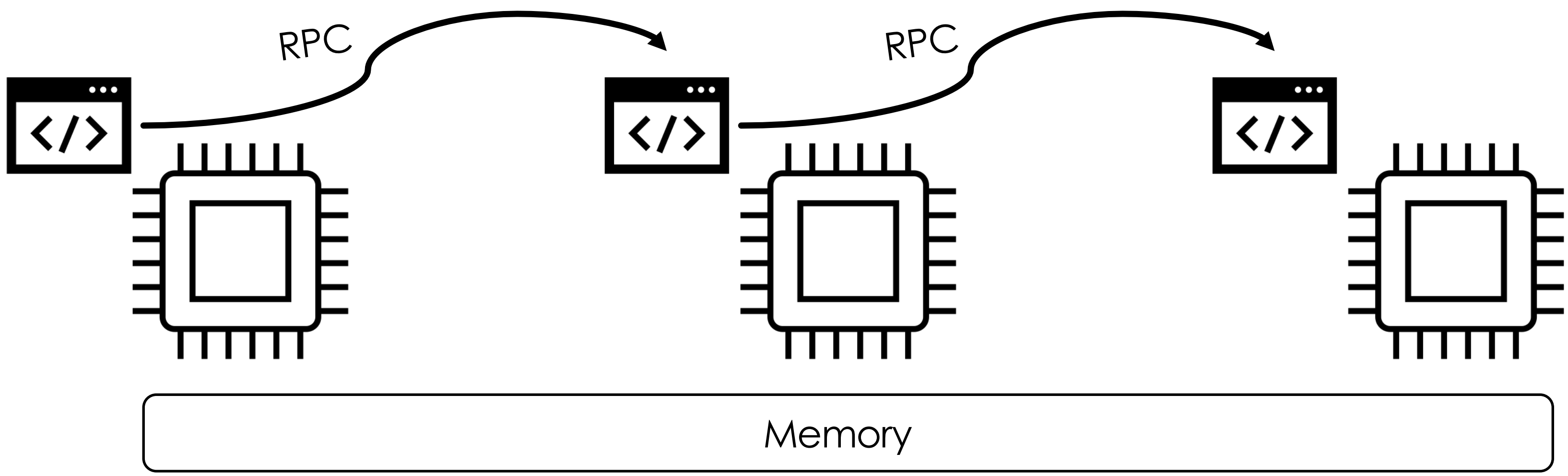
- **Even highly specialized software context switching penalty not negligible**

We need a specialized hardware support for context switching



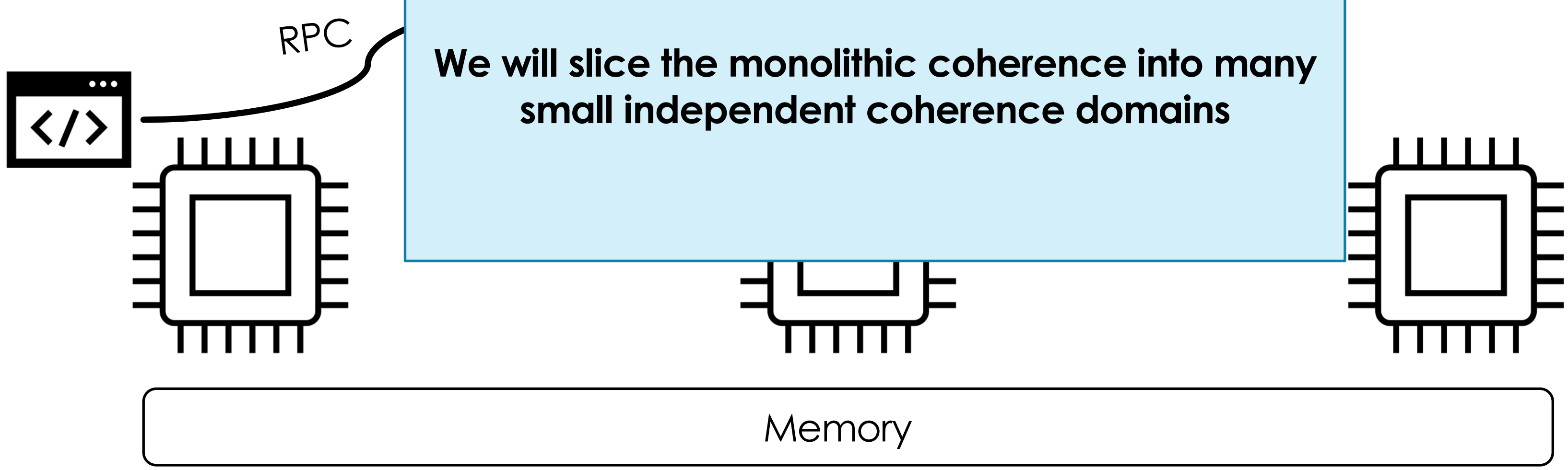
Is chip-wide monolithic cache coherence needed?

- Services use RPCs for the communication, no shared memory



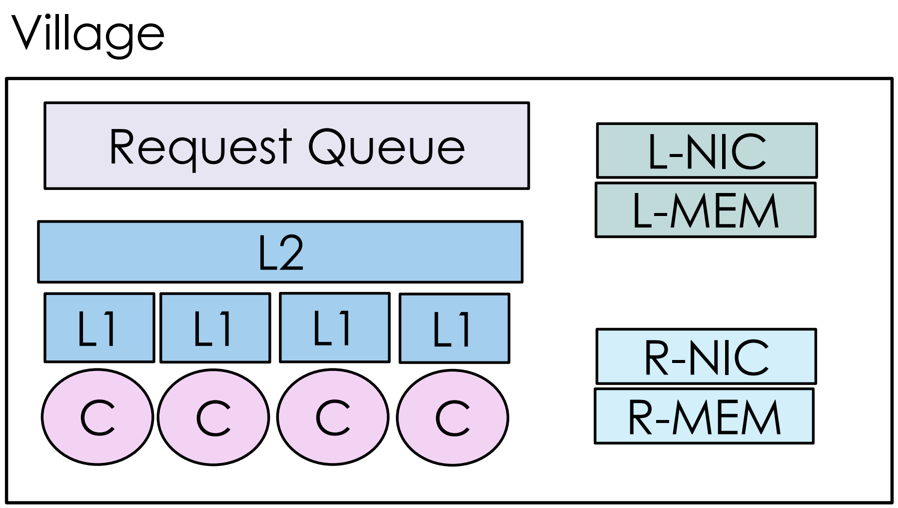
Is chip-wide monolithic cache coherence needed?

- Services use RPCs for the communication, no shared memory



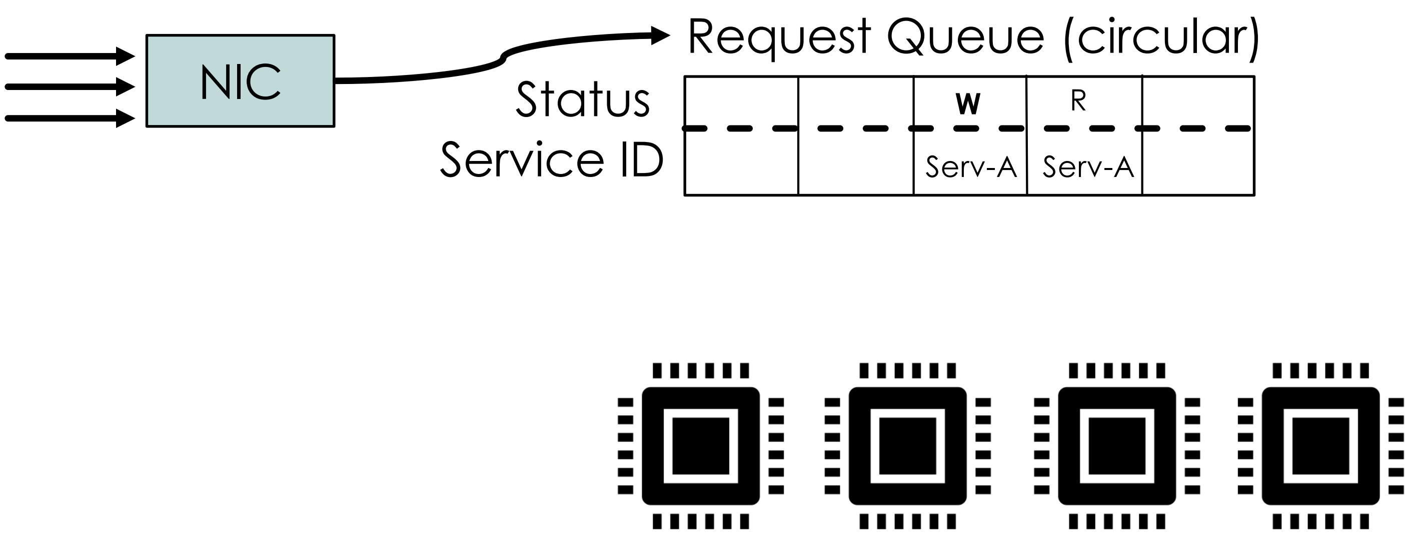
We will slice the monolithic coherence into many small independent coherence domains

Basic unit of µManycore: a hardware cache-coherent Village



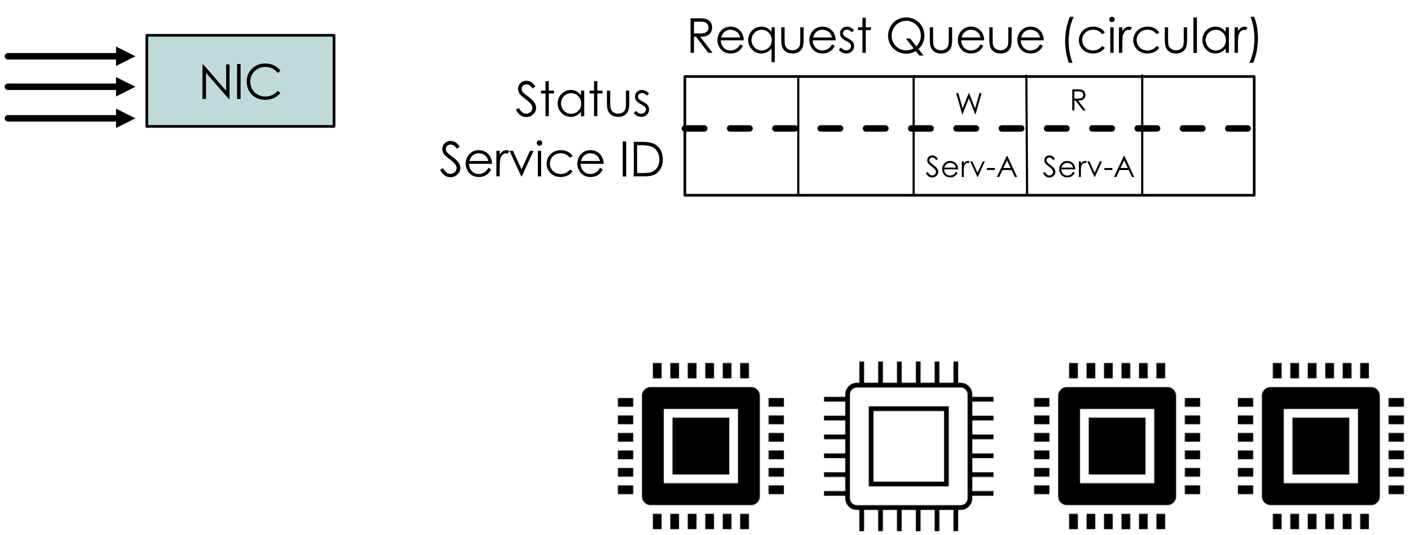
Hardware for Request Scheduling

- NIC deposits ready requests to the queue
- Cores spin on *Work* flag, execute *Dequeue* instruction, finish with *Complete* instruction



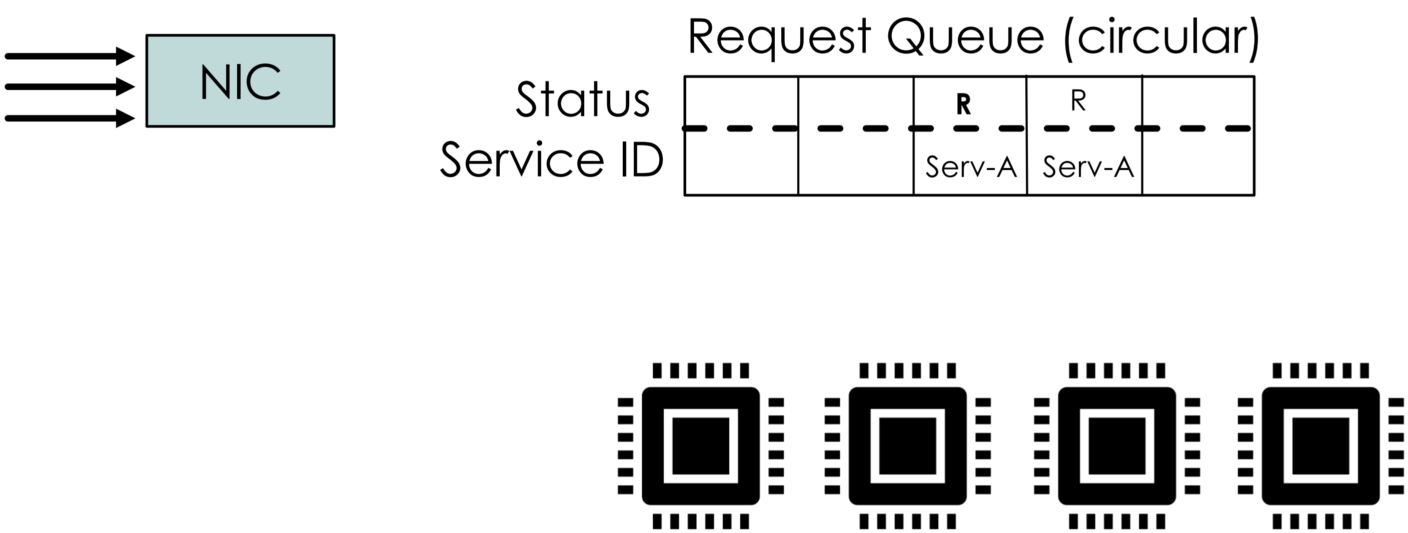
Hardware for Request Scheduling

- NIC deposits ready requests to the queue
- Cores spin on *Work* flag, execute *Dequeue* instruction, finish with *Complete* instruction



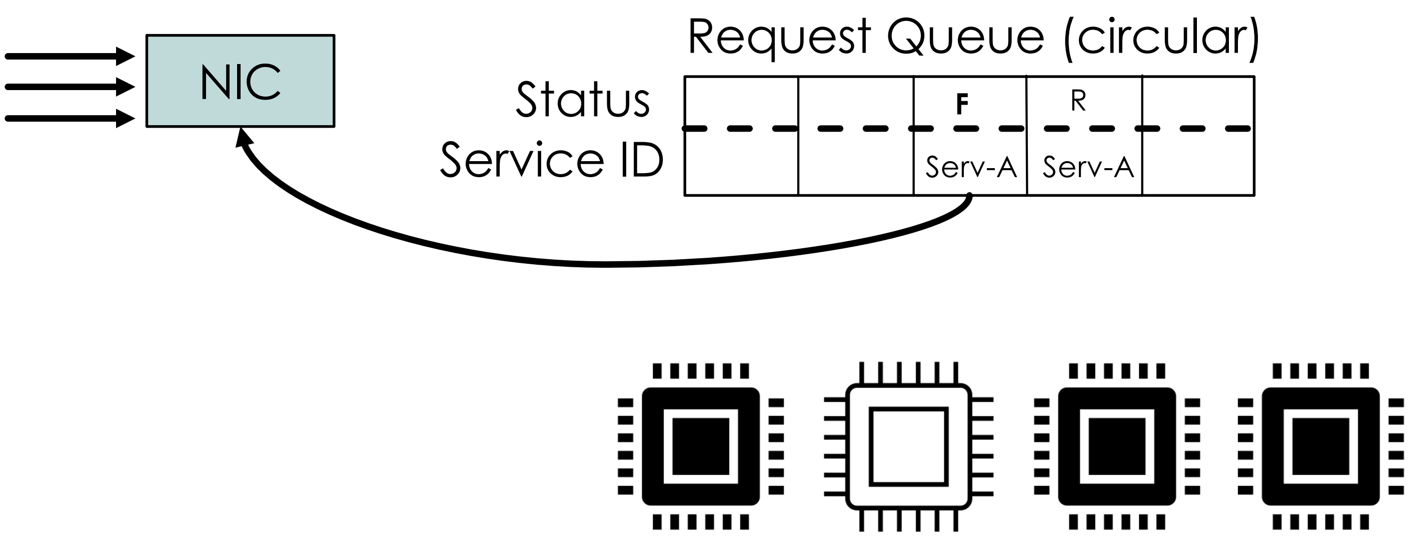
Hardware for Request Scheduling

- NIC deposits ready requests to the queue
- Cores spin on *Work* flag, execute *Dequeue* instruction, finish with *Complete* instruction



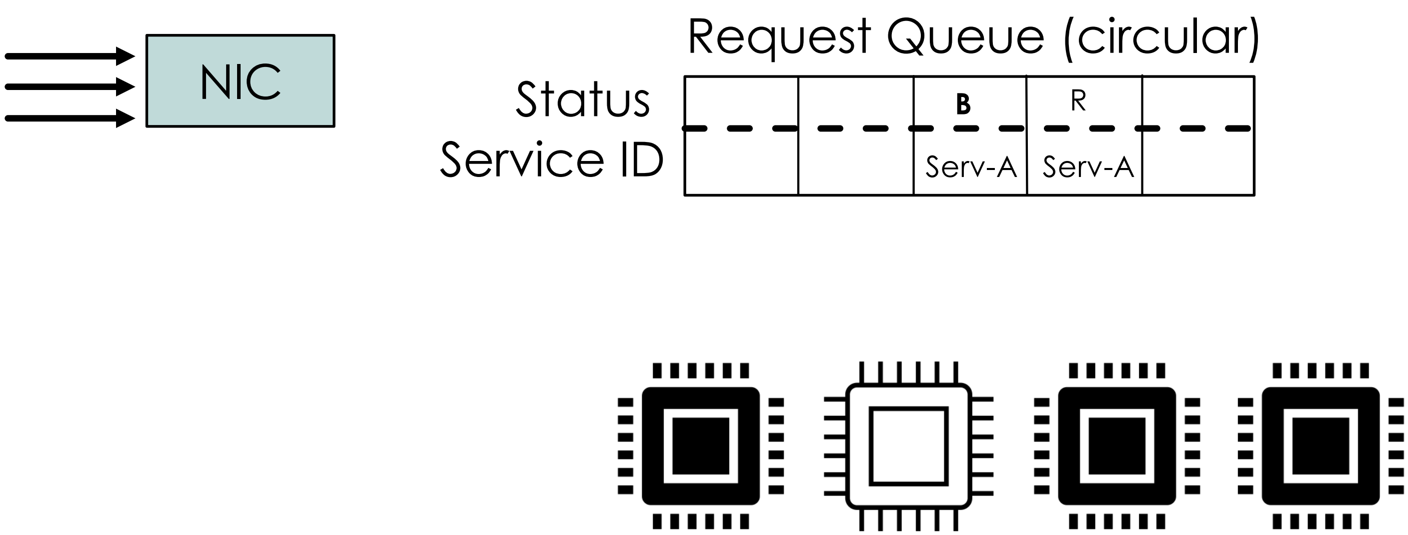
Hardware for Request Scheduling

- NIC deposits ready requests to the queue
- Cores spin on *Work* flag, execute *Dequeue* instruction, finish with *Complete* instruction



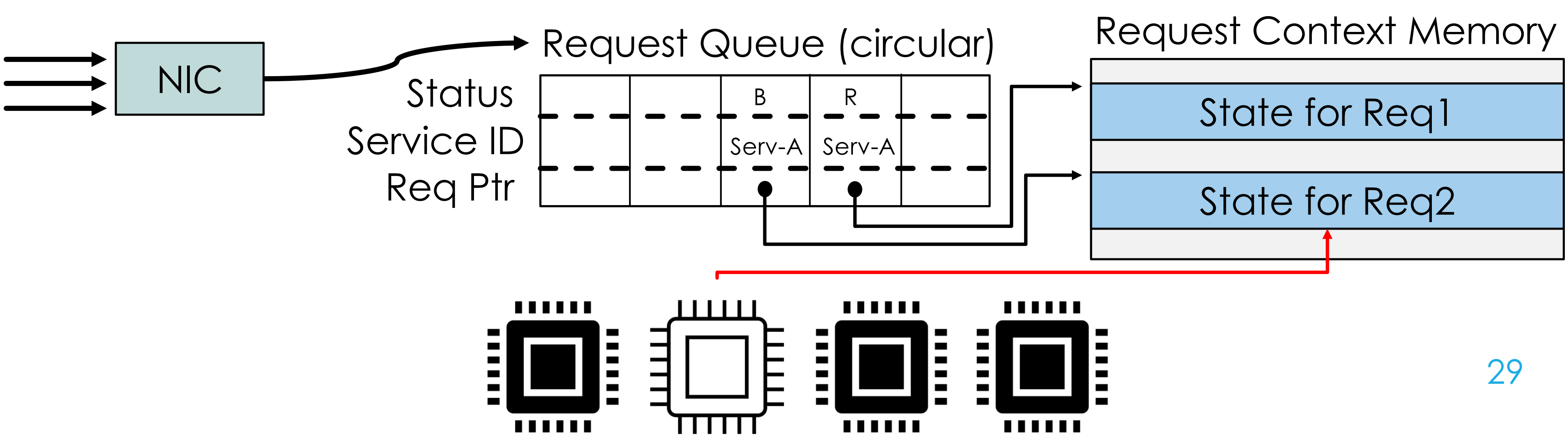
Hardware for Context Switching

- Requests can get blocked during execution – need to context switch



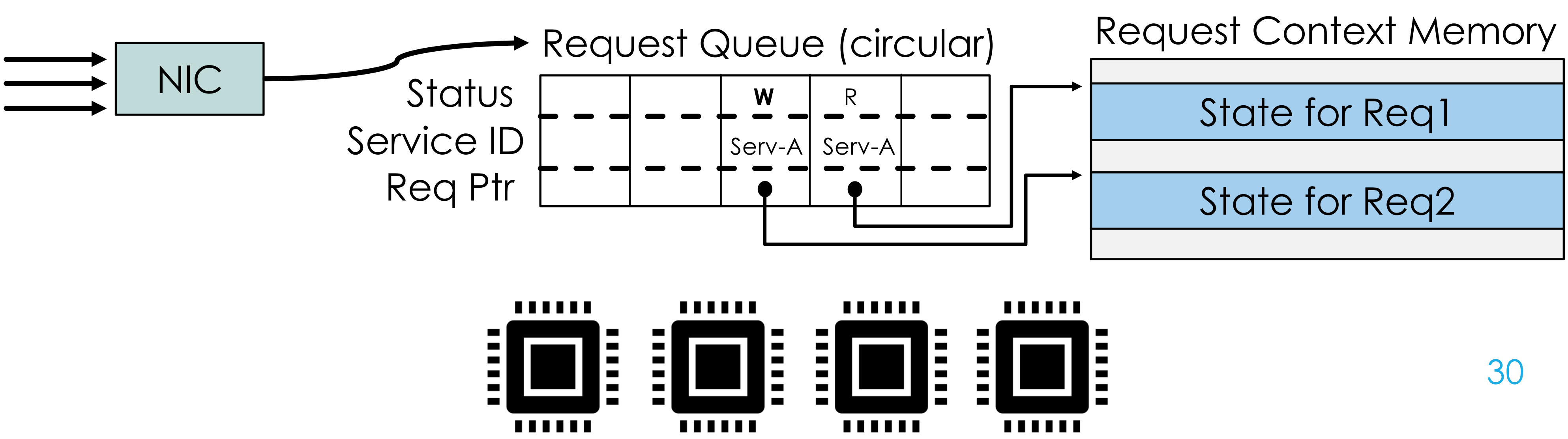
Hardware for Context Switching

- Avoid OS invocations and software overheads
- Core saves and restores context in hardware



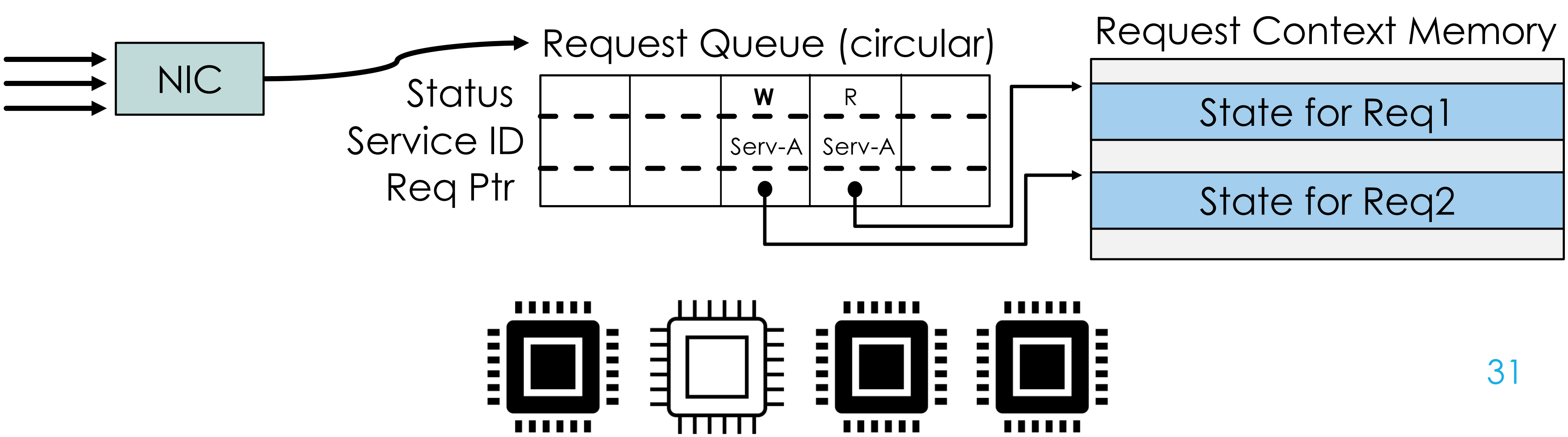
Hardware for Context Switching

- Avoid OS invocations and software overheads
- Core saves and restores context in hardware



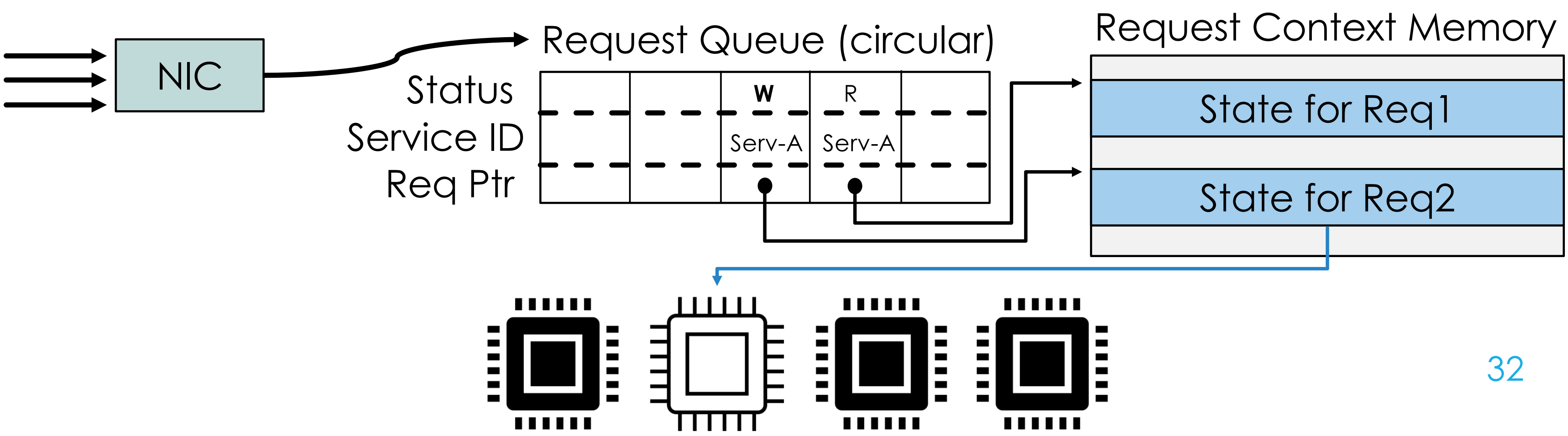
Hardware for Context Switching

- Avoid OS invocations and software overheads
- Core saves and restores context in hardware



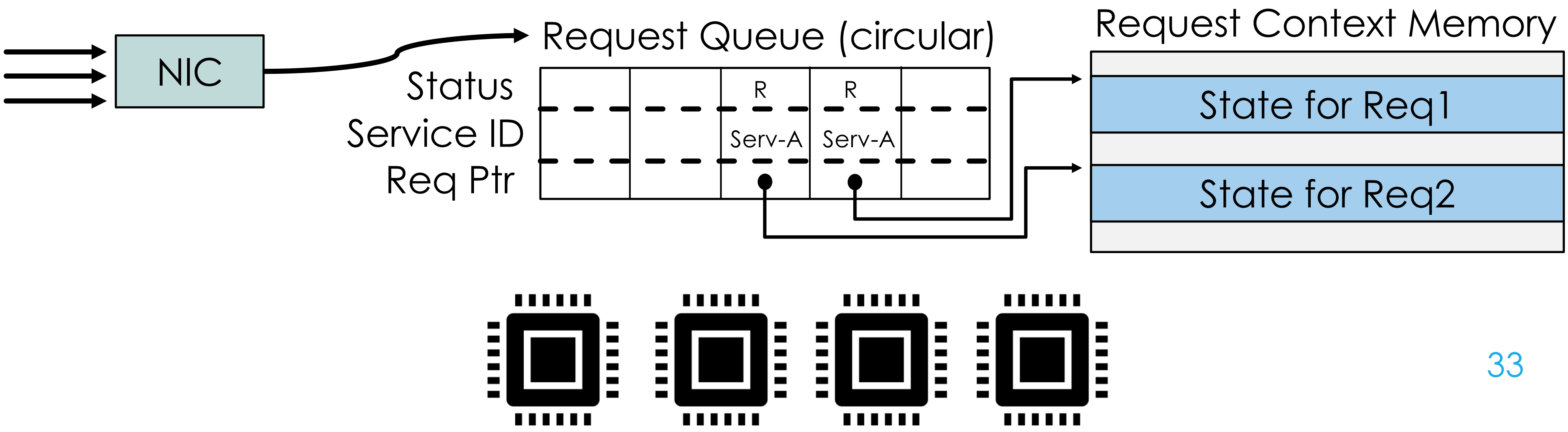
Hardware for Context Switching

- Avoid OS invocations and software overheads
- Core saves and restores context in hardware



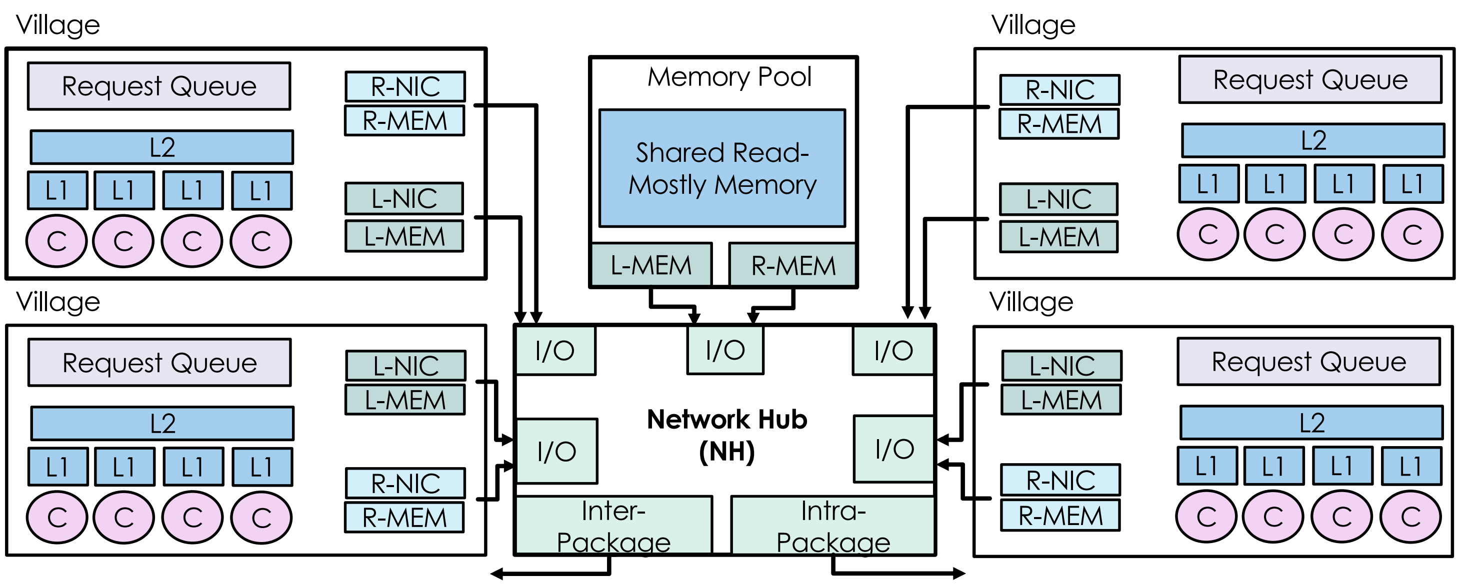
Hardware for Context Switching

- Avoid OS invocations and software overheads
- Core saves and restores context in hardware



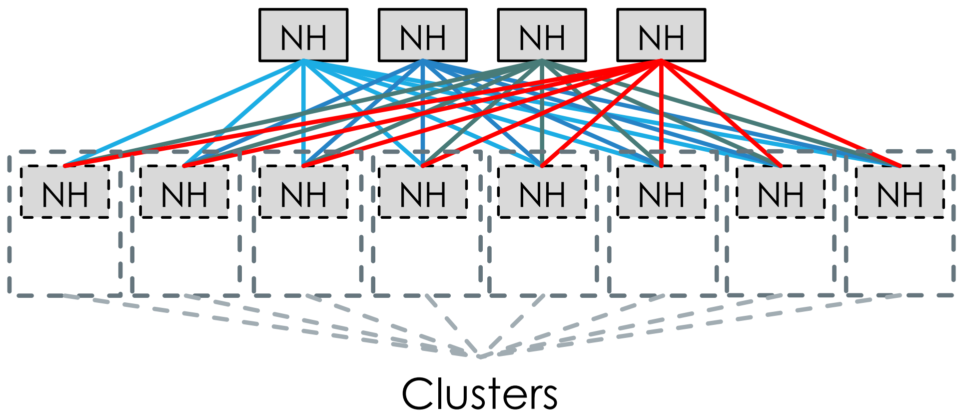
Villages grouped into clusters

- The combination of a few villages, a memory pool, and a network hub → a cluster



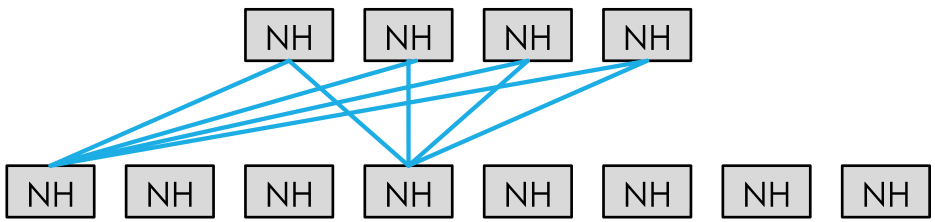
Leaf-spine on-package network

- Many redundant, low-hop count paths between any two clusters



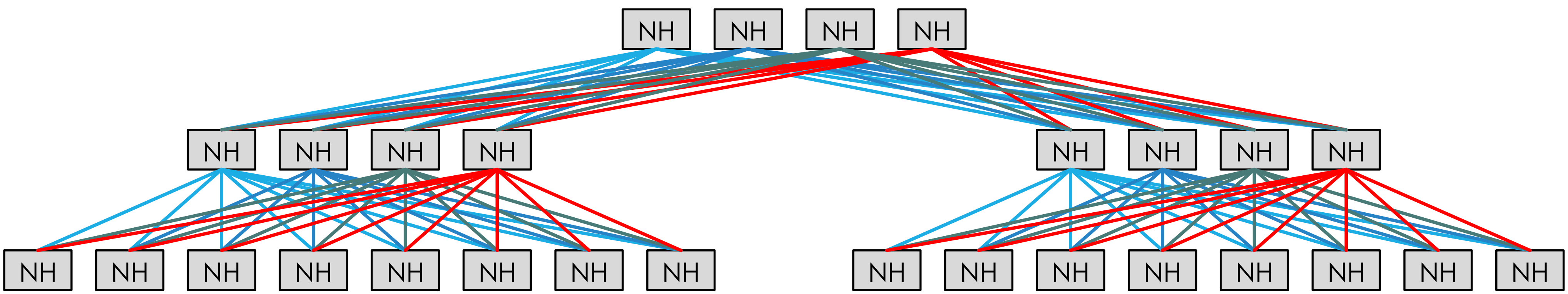
Leaf-spine on-package network

- Many redundant, low-hop count paths between any two clusters
  - Even between the same source and destination multiple parallel links



Hierarchical leaf-spine on-package network

- Many redundant, low-hop count paths between any two clusters

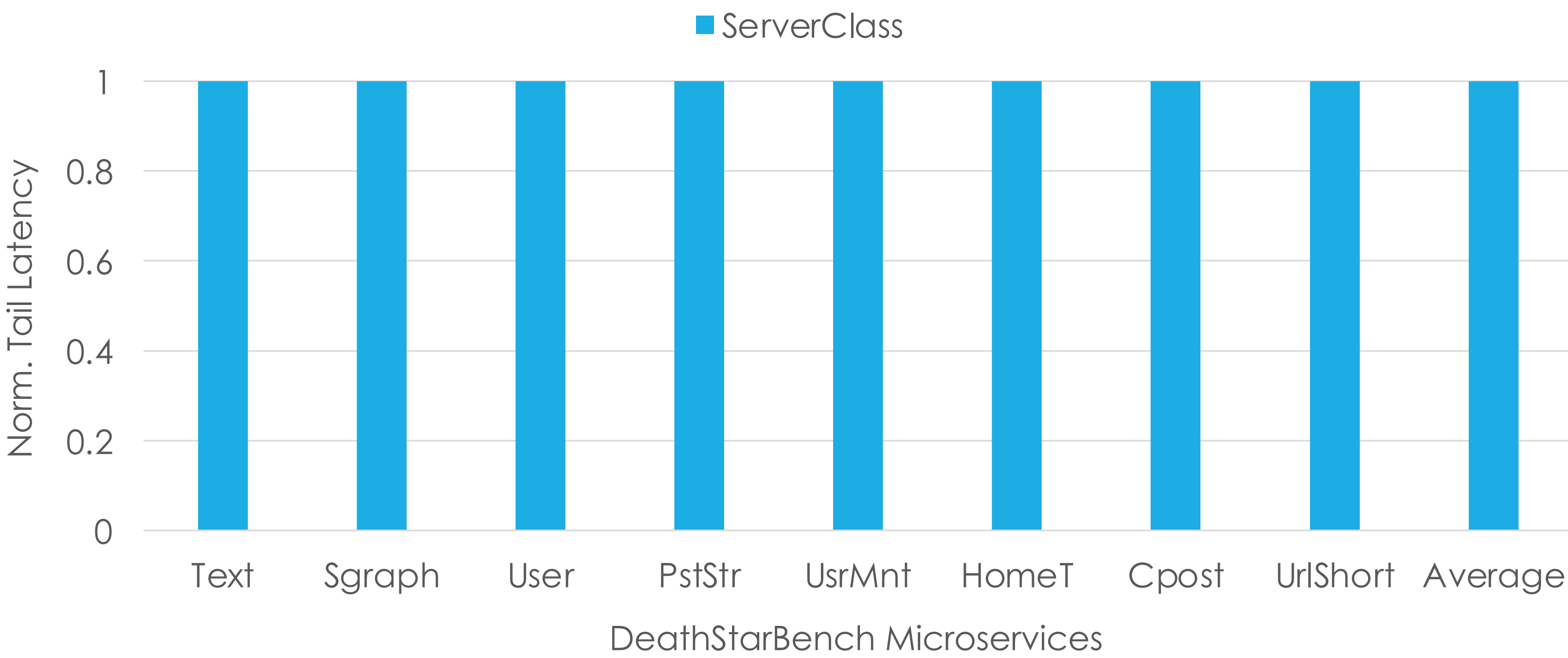


Evaluation Setup

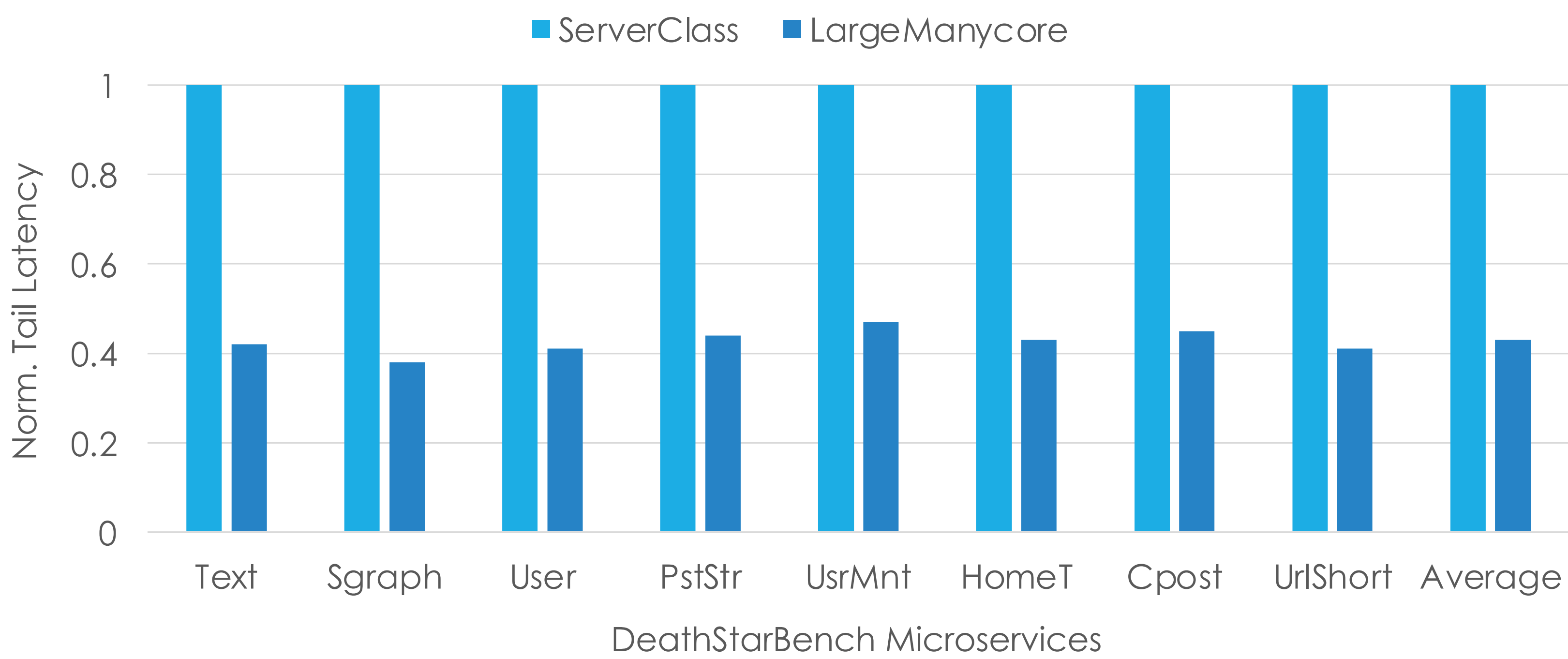
- 1024-core µManycore
- DeathStarBench microservices
- PinTool to extract traces
- SST for cycle-accurate timing measurements
- McPAT + Cacti for power/area measurements
- Two baselines

Baseline	Number of cores	Modeled After	Design Point
ServerClass	40	Intel Ice-Lake	Same Power as µManycore
LargeManycore	1024	ARM A15	Same Area as µManycore

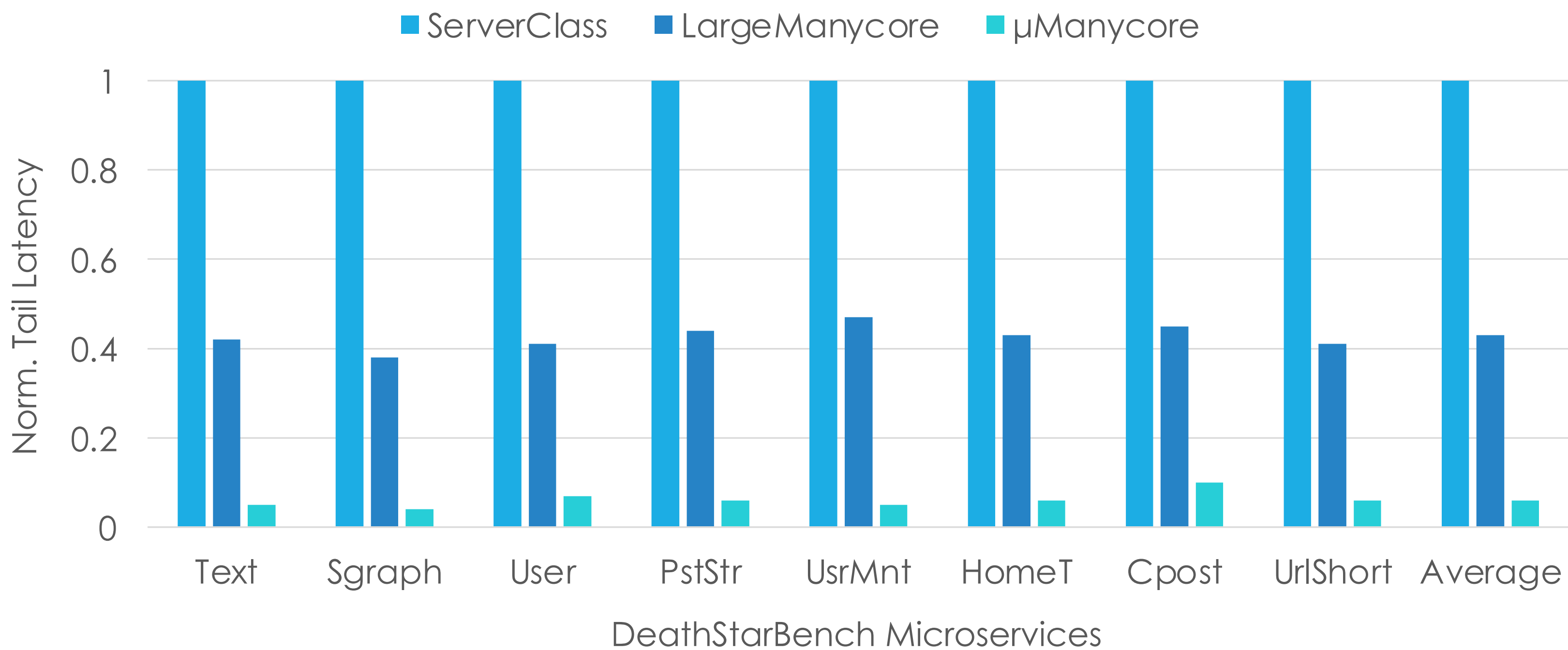
µManycore Significantly Reduces Tail



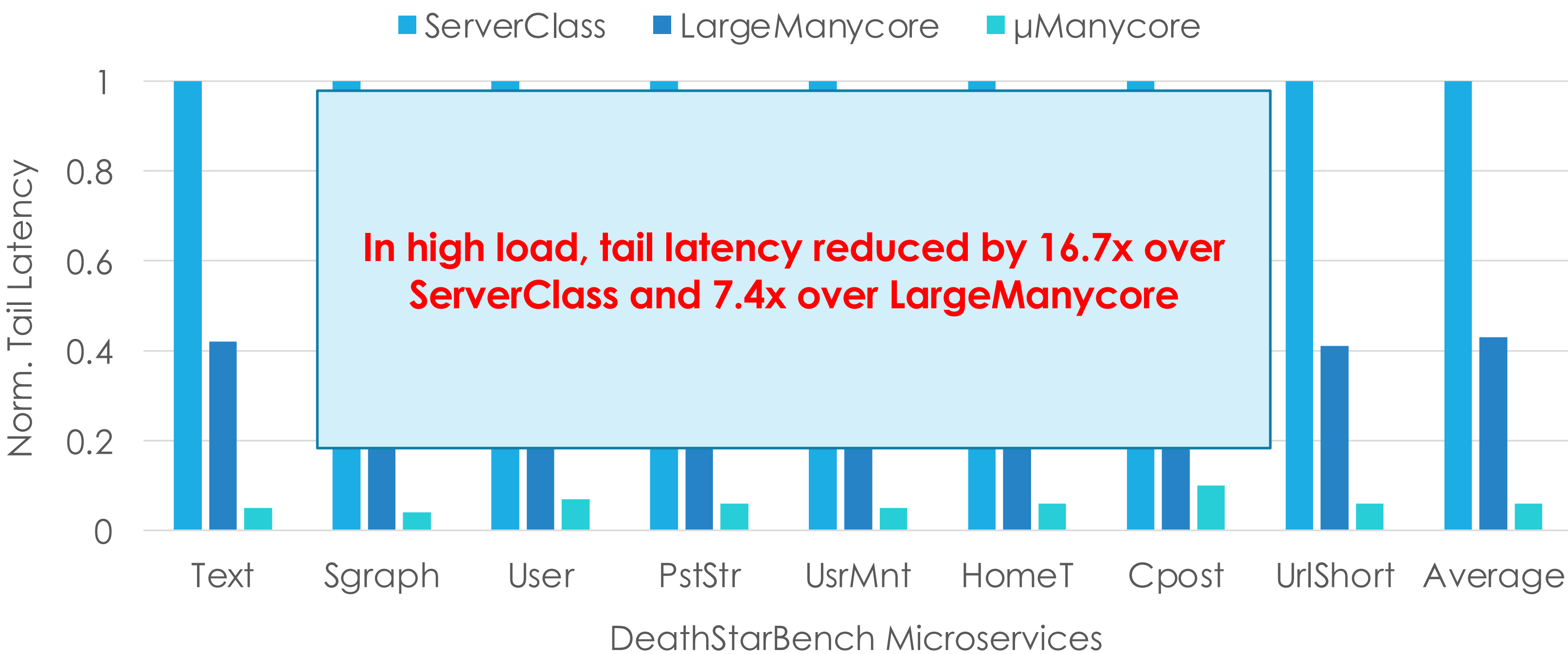
µManycore Significantly Reduces Tail



µManycore Significantly Reduces Tail



µManycore Significantly Reduces Tail



In high load, tail latency reduced by 16.7x over ServerClass and 7.4x over LargeManycore

Conclusion

- Imbalance between current processors and emerging microservice environments
- μManycore → an architecture optimized for microservice environments
- μManycore delivers high performance for microservice workloads
  - 10.4X reduced tail latency
  - 15.5X improved throughput

µManycore:
A Cloud-Native CPU for Tail at Scale

ISCA 2023

Jovan Stojkovic, Chunao Liu*, Muhammad Shahbaz*, Josep Torrellas
University of Illinois at Urbana-Champaign, *Purdue University

Simulation Parameters

ScaleOut == LargeManycore

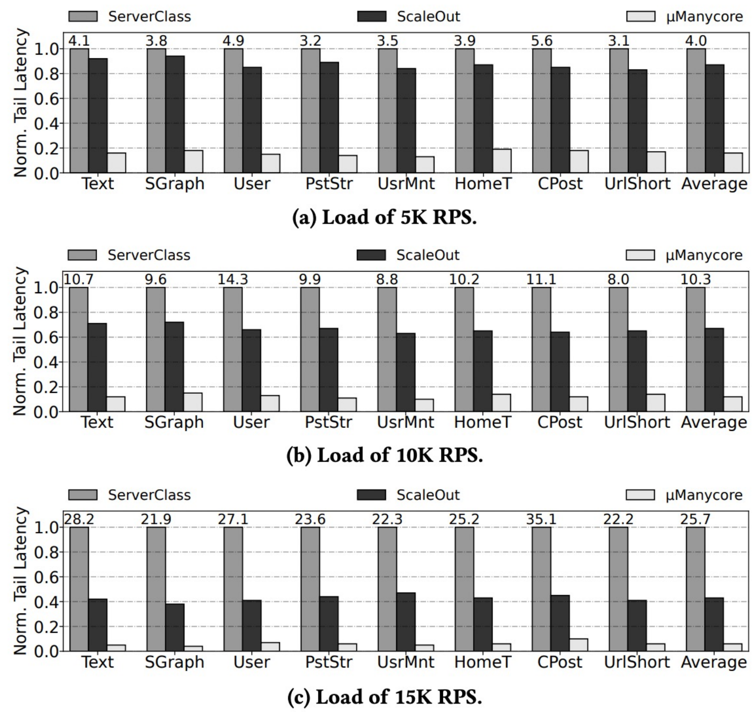
Table 2: Architectural parameters used in the evaluation.

ServerClass Multicore	
Multicore	40 (or 128) 6-issue cores, 352-entry ROB, 256-entry LSQ, 3GHz
L1 cache	64KB, 8-way, 2 cycles round trip (RT), 64B line
L2 cache	2MB, 16-way, 16 cycles RT, 20 MSHRs
L3 cache	2MB/core, 16-way, 40 cycles RT, 20 MSHRs
L1 DTLB	256 entries, 4-way, 2 cycles RT
L2 DTLB	2048 entries, 12-way, 12 cycles RT
Network	2D mesh
μManycore and *ScaleOut* Manycores	
Manycore	1024 4-issue cores, 64-entry ROB, 64-entry LSQ, 2GHz
L1 cache	64KB, 8-way, 2 cycles RT, 64B line
L2 cache	256KB, 16-way, 24 cycles RT, 20 MSHRs
L1 DTLB	128 entries, 4-way, 2 cycles RT
Network	Fat tree (*ScaleOut*), leaf-spine (*μManycore*)
Network	
Intra server	5 cycles/hop (4 router delay + 1 wire delay) [9]
Inter server	1μs RT; 200GB/s
Main-memory per Server	
Capacity	80GB
Channels; Banks	4; 8
Frequency; Rate	1GHz; DDR
Mem bandwidth	8 memory controllers; 102.4GB/s per controller

Tail Latency with Different Loads

On average, μManycore reduces the tail latency

over ServerClass by 6.3×, 8.3×, and 16.7
over ScaleOut by 5.4×, 6.5×, and 7.4×



(a) Load of 5K RPS.

(b) Load of 10K RPS.

(c) Load of 15K RPS.

Figure 14: Tail latency in *ServerClass*, *ScaleOut*, and *µManycore* normalized to *ServerClass*. The numbers on top of the *ServerClass* bars are the absolute latency values in ms.

Tail Latency Breakdown

On average, the cumulative application of these techniques reduces the tail latency by 1.1×, 2.3×, 3.9×, and 7.4×, respectively

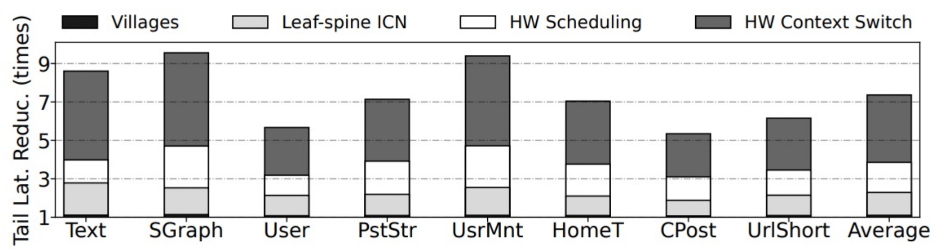


Figure 15: Contributions of the four main *μManycore* techniques to the reduction of tail latency for 15K RPS. Latency reductions are normalized to the tail latency of *ScaleOut*.

Average Latency with Different Loads

On average, μManycore reduces the average latency over ServerClass by 2.3×, 3.2×, and 5.6× for loads of 5K, 10K, and 15K RPS, respectively, and over ScaleOut by 2.1×, 2.5×, and 3.2× for the same loads

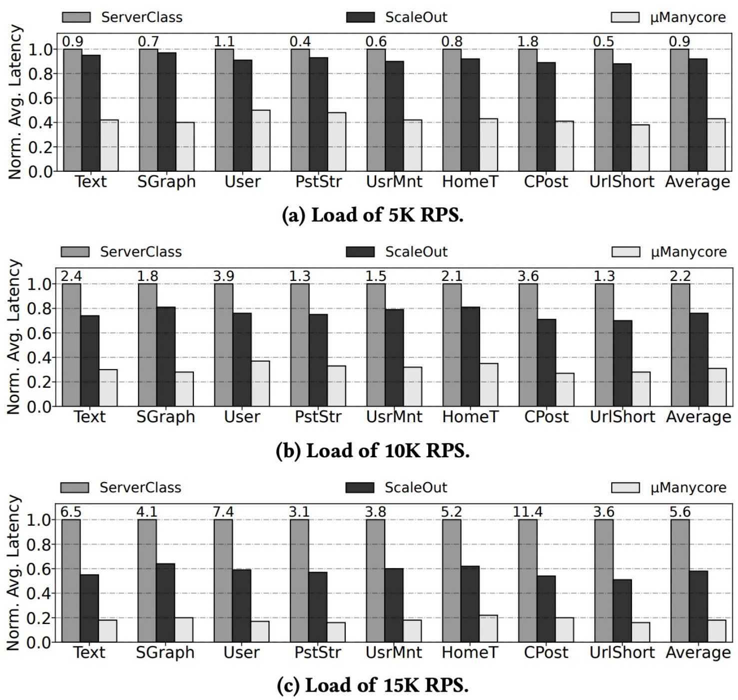


Figure 16: Average latency in *ServerClass*, *ScaleOut*, and *μManycore* normalized to *ServerClass*. The numbers on top of the *ServerClass* bars are the absolute latency values in ms.

Average Latency with Different Loads

μManycore reaches (… n
average, μManycore … lass and
ScaleOut baselines, r…

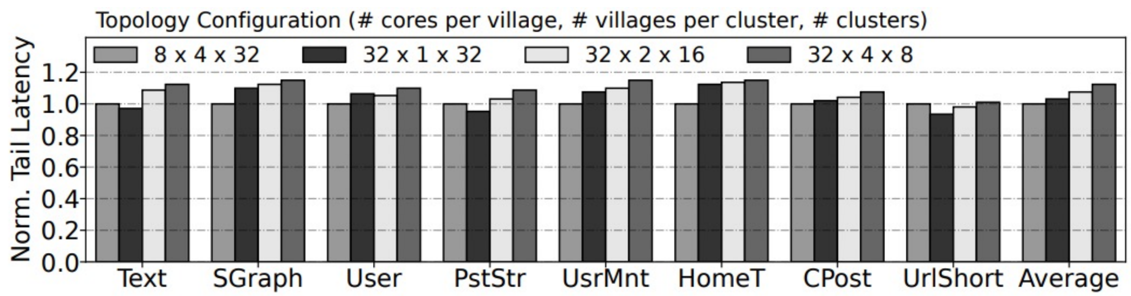


Figure 19: Normalized tail latency with different *μManycore* configurations.

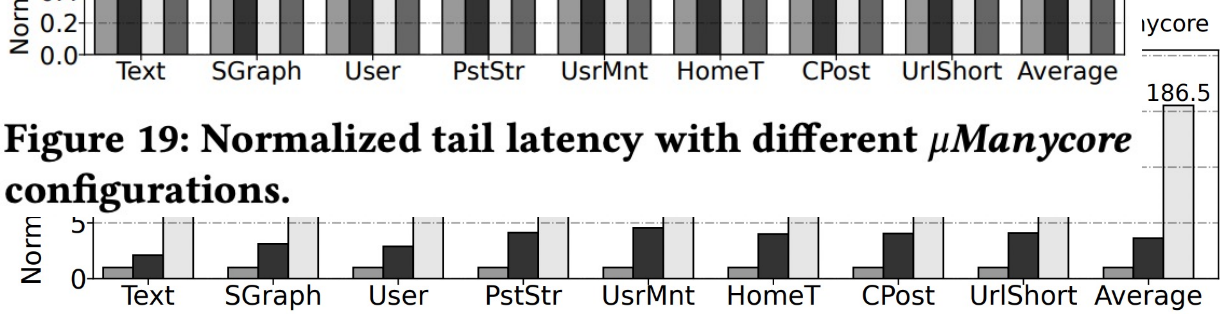


Figure 18: Normalized maximum throughput a system can achieve without violating QoS guarantees. The numbers on top of the *μManycore* bars are the absolute throughput values that *μManycore* achieves.

Sensitivity Study on Village Sizes

All configurations are within 15% of each other's tail latency

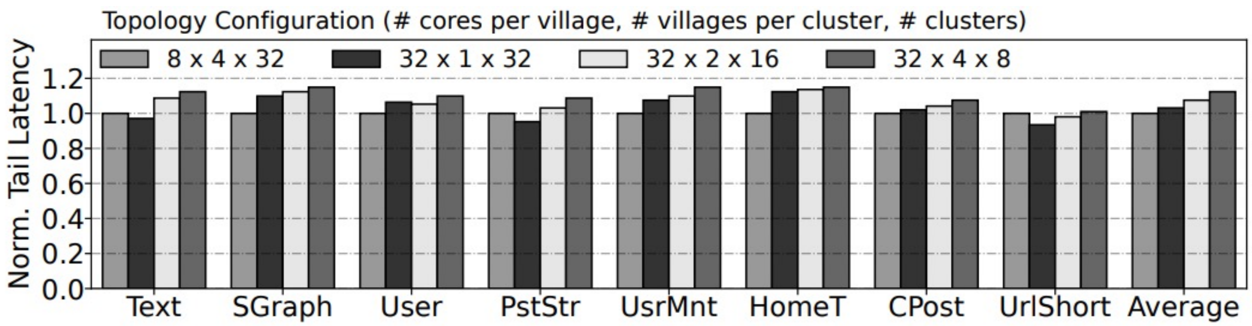


Figure 19: Normalized tail latency with different *μManycore* configurations.

Iso-area ServerClass Baseline

- In the iso-power configurations, μManycore has 2.9% more area than ScaleOut and 3.1× more area than the 40-core ServerClass (i.e., $547.2mm2$ for μManycore versus $176.1mm2$ for ServerClass)
- For an iso-area comparison, we keep μManycore and ScaleOut unchanged and we scale ServerClass to 128 cores, while leaving all the other parameters unmodified
- ServerClass processor improves the performance significantly, matching and sometimes slightly outperforming the tail latency of ScaleOut
- ServerClass still has a tail latency that is on average 7.3× higher than the μManycore one across all loads and applications
- Also, the 128-core ServerClass processor uses an unacceptably large amount of power, namely 3.2× more than μManycore.