Enhancing Server Efficiency in the Face of **Killer Microseconds**

Amirhossein Mirhosseini, Akshitha Sriraman, Thomas F. Wenisch

University of Michigan

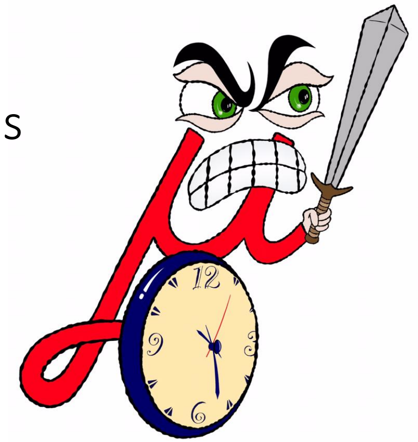
HPCA 2019

02 /18/2019

Killer Microseconds

[Barroso'17]

- Frequent microsecond-scale pauses in datacenter applications
  - Stalls for accessing emerging memory & I/O devices
  - Mid-tier servers synchronously waiting for leaf nodes
  - Brief idle periods in high-throughput microservices



- Modern computing systems not effective in hiding microseconds
  - Micro-architectural techniques are insufficient
  - OS/software context switches are too coarse grain

Our proposal: **Duplexity**

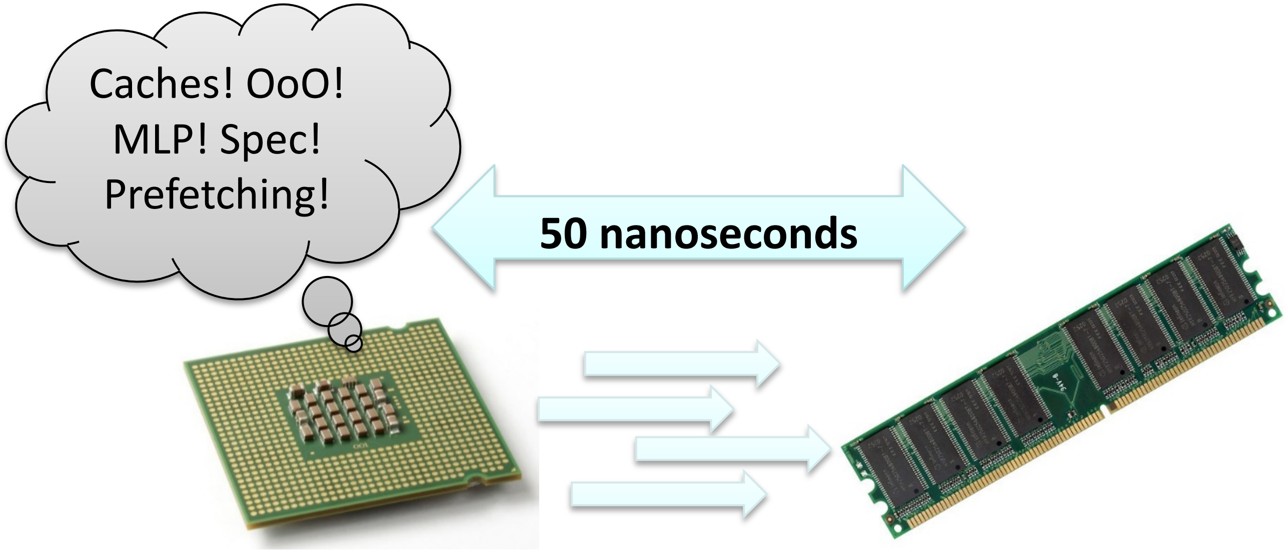
- Cost-effective highly multithreaded server design
- Heterogeneous design --- ***Dyads*** of cores:
  - ***Master core*** for latency-sensitive microservices
  - ***Lender core*** for latency-insensitive applications
- ***Key idea 1:*** master core may "borrow" threads from the lender core to fill utilization holes
- ***Key idea 2:*** cores protect threads' cache states to avoid excessive tail latencies and QoS violations

Duplexity improves core utilization by 4.8x in presence of killer microseconds

Outline

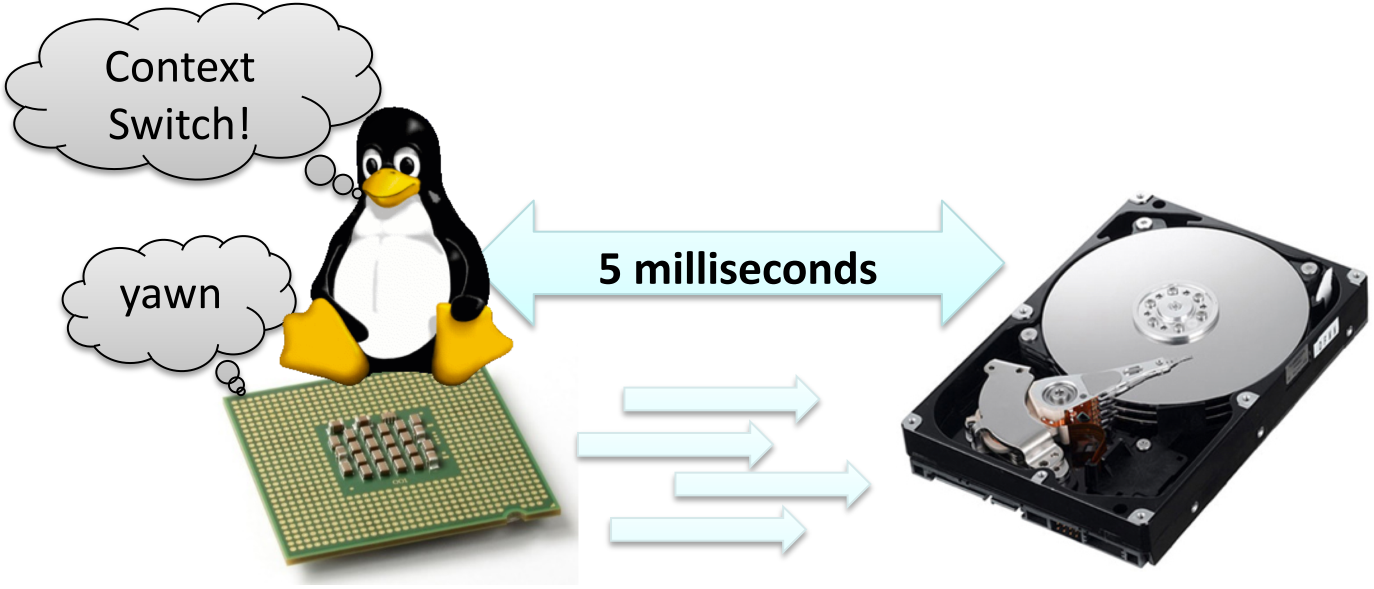
- Killer microseconds
- Why (scaling) SMT is not an option
- Duplexity server architecture
- Evaluation methodology and results

Modern HW is great at hiding nanosecond scale stalls…



Micro-architectural techniques are at best able to hide 100s of nanoseconds

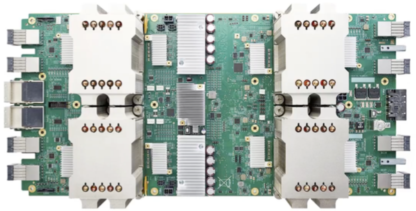
Modern OS is great at hiding millisecond scale stalls…



OS context switching typically has an average overhead of at least 5-20us

But, today's devices and microservices inflict μs-scale stalls

- ***Emerging memory and I/O technologies:***
  - NVM, disaggregated memory, … : O(1 μs)
  - High-end flash, accelerators, … : O(10 μs)



- ***Brief idle periods:***
  - With μs-scale microservices, idle periods also shrink to μs scales
    - 200K QPS service at 50% load has average idle periods of only 10 μs

Need HW/SW mechanisms to hide μs-scale latencies

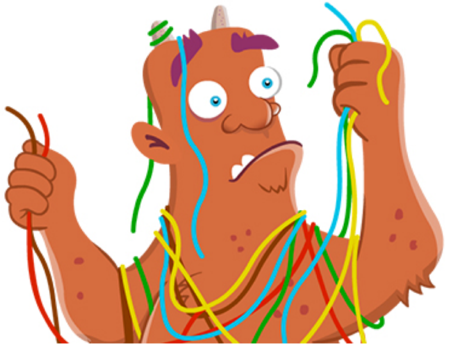
Multithreading is the obvious solution

- OS context switches are too coarse-grain for µs-scale periods
  - User-level cooperative multithreading [Cho'18]
  - Hardware (simultaneous) multithreading [Yamamoto'95][[Tullsen'95][Tullsen'96] …

But, we need a lot of (10+) threads to fill µs-scale stall/idle periods

Simply adding more threads is not enough

- Complicates fetch/dispatch/issue logic
  - Prolonging its critical path
- Requires a larger register file
- Pressure/thrashing in L1 caches
- **Higher tail latency due to interference among threads**
  - Up to 5.7x higher tail latency
  - 1.5x higher tail at low load and low IPC co-runner



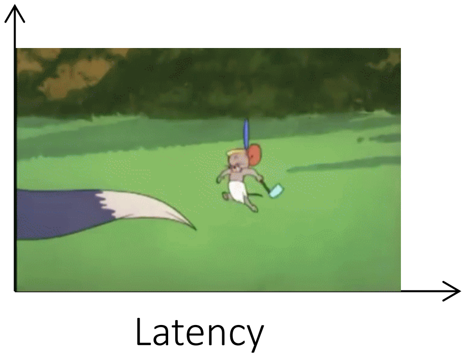
Need ***complexity management*** and ***performance isolation*** mechanisms

Duplexity

- Two main objectives:
  - Maximize performance density and energy efficiency
    - Fill utilization "holes" arising from killer microseconds

Borrow latency-insensitive batch threads to fill microservices' ***utilization holes***

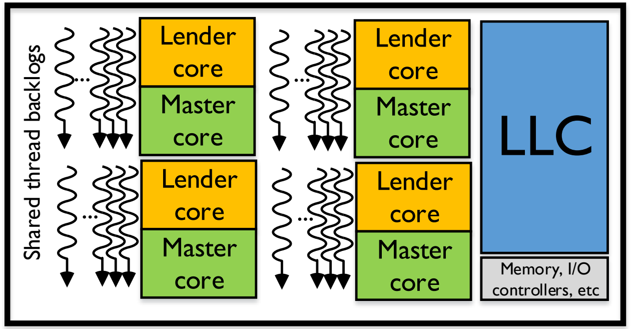
  - Minimize disruption of latency-sensitive threads
    - Avoid excessive tail latency due to interference



Isolate stateful uarch structures (e.g., caches) to avoid ***QoS violations***

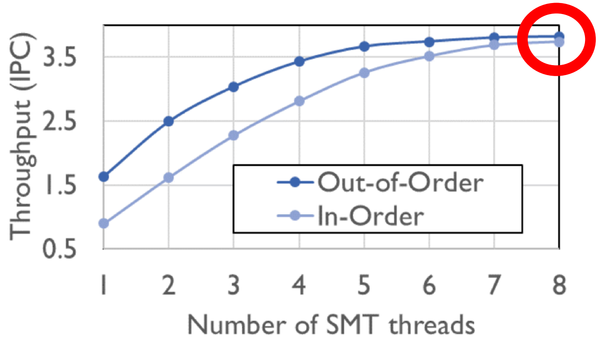
Duplexity : a server made of "Dyads"

- Master core
  - Designed for latency-sensitive microservices
  - "Borrows" threads from lender core to fill util holes
- Lender core
  - Designed for latency-insensitive batch applications
- Shared backlog of batch threads

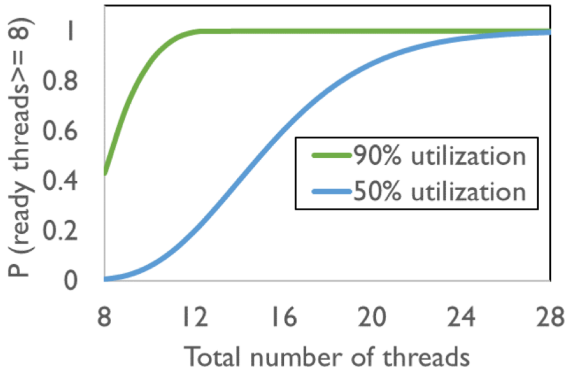


Lender Core

- Latency-insensitive batch threads
  - In-order execution



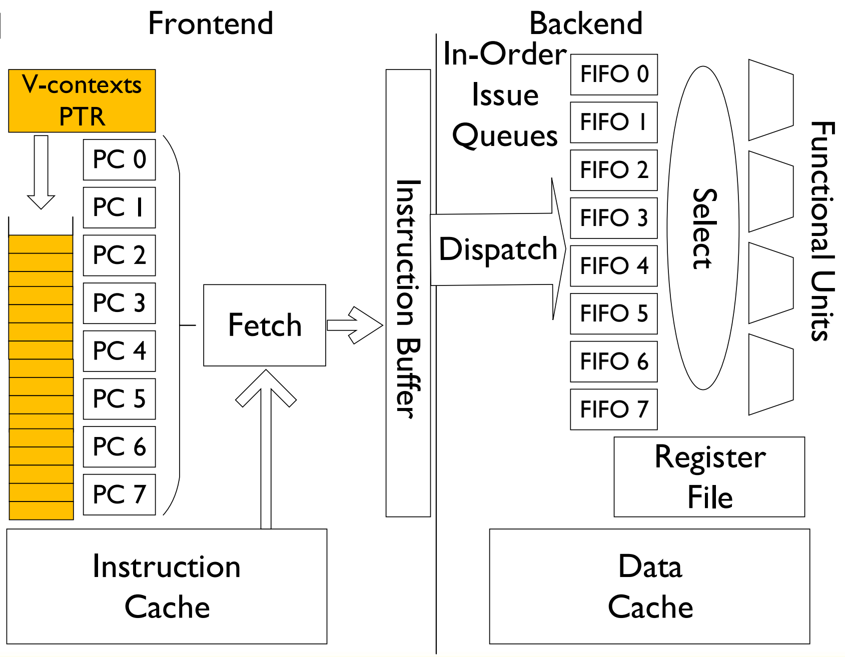
- Variable number of virtual contexts needed
  - FIFO run-queue of virtual contexts in memory



Lender Core

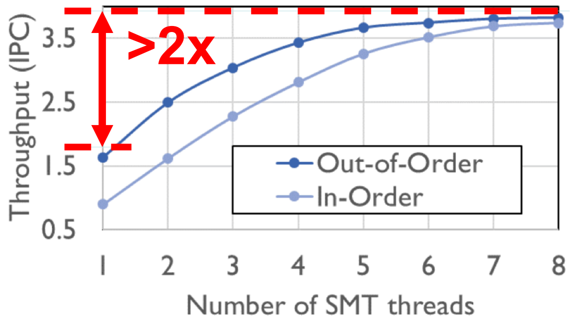
- Hierarchical Simultaneous Multithreading (HSMT)
  - Backlog of virtual contexts
  - Inspired by Balanced Multithreading [Tune'04]

8-way In-order SMT Datapath



Master Core

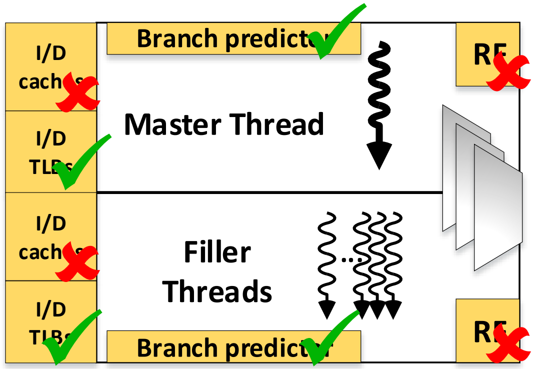
- Single latency-sensitive **master** thread
- Borrows threads from the lender core to fill µs-scale holes
  - Single-threaded out-of-order mode for ***master*** thread
  - Multi-threaded in-order mode for ***filler*** threads
- Inspired by Morphcore [Khubaib'12]



Filler threads thrash the cache, TLB, and branch predictor state of the master thread
➔ Increase tail latency

Segregating State

- **Naive solution**: replicate all stateful uarch structures
  - Register files, caches, branch predictor, TLBs, etc.

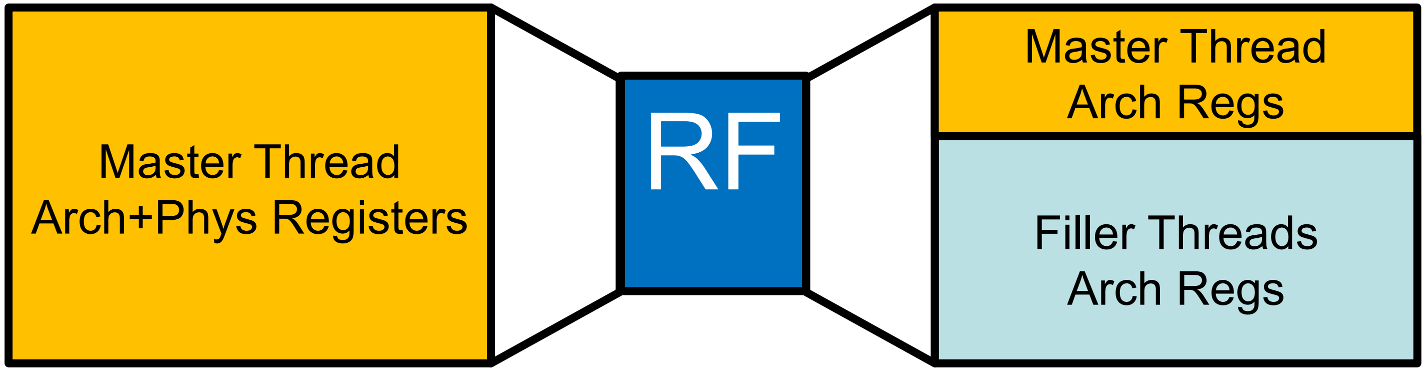


Master core only replicates **inexpensive structures** (e.g., TLBs and predictors)

Caches and register files are large and power-hungry
➔ Full replication undermines performance density and energy efficiency objectives

Segregating Register Files

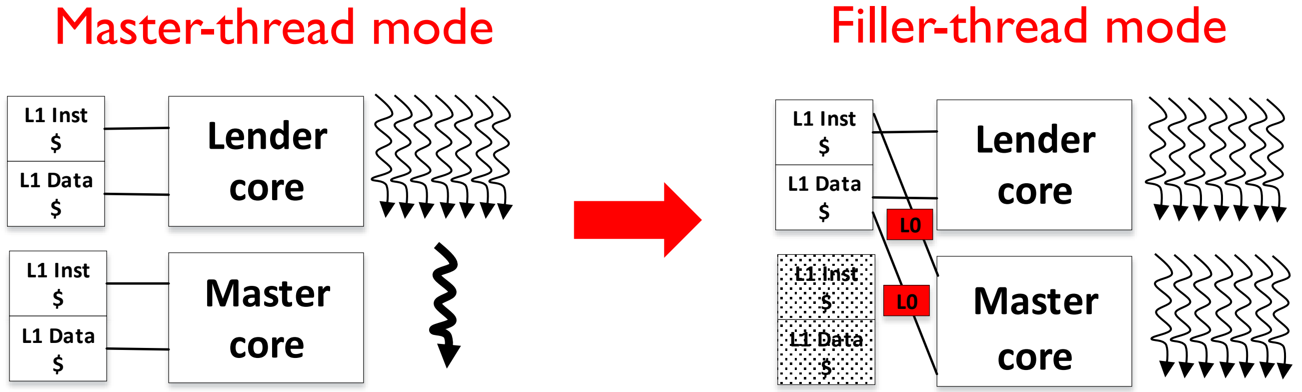
- Repurpose physical RF as architectural RF for filler threads
- Retain master thread architectural registers
  - Facilitates fast restart when the stall resolves



What about caches?

Master-Lender Dyads

- Master core remotely accesses the L1 I/D caches of the lender core
  - Protects the master thread's state
  - Allows filler threads to hit on their own cache state
- L0 I/D caches as effective bandwidth filters

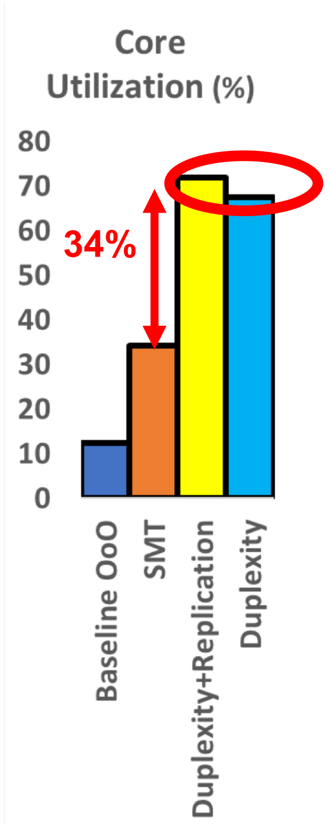


Master thread can almost immediately resume execution as stall resolves

Evaluation Methodology

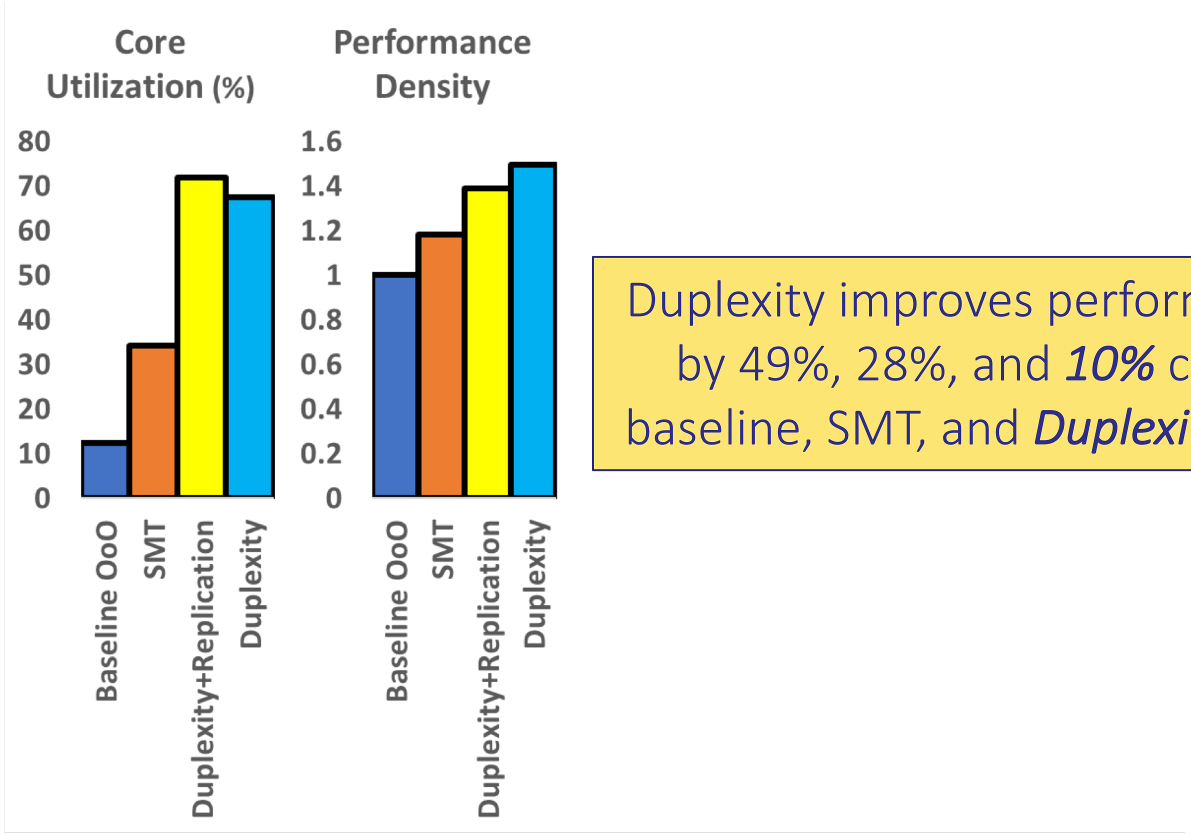
- Master thread:
  - Open source µs-scale microservices
    - Locality sensitive hashing, protocol routing, remote caching, word stemming
- Filler threads:
  - Data-parallel distributed graph algorithms
    - Page Rank, single-source shortest path
- Design Alternatives:
  - Baseline single-threaded OoO, SMT, Duplexity+Replication, more alternatives in the paper

Evaluation



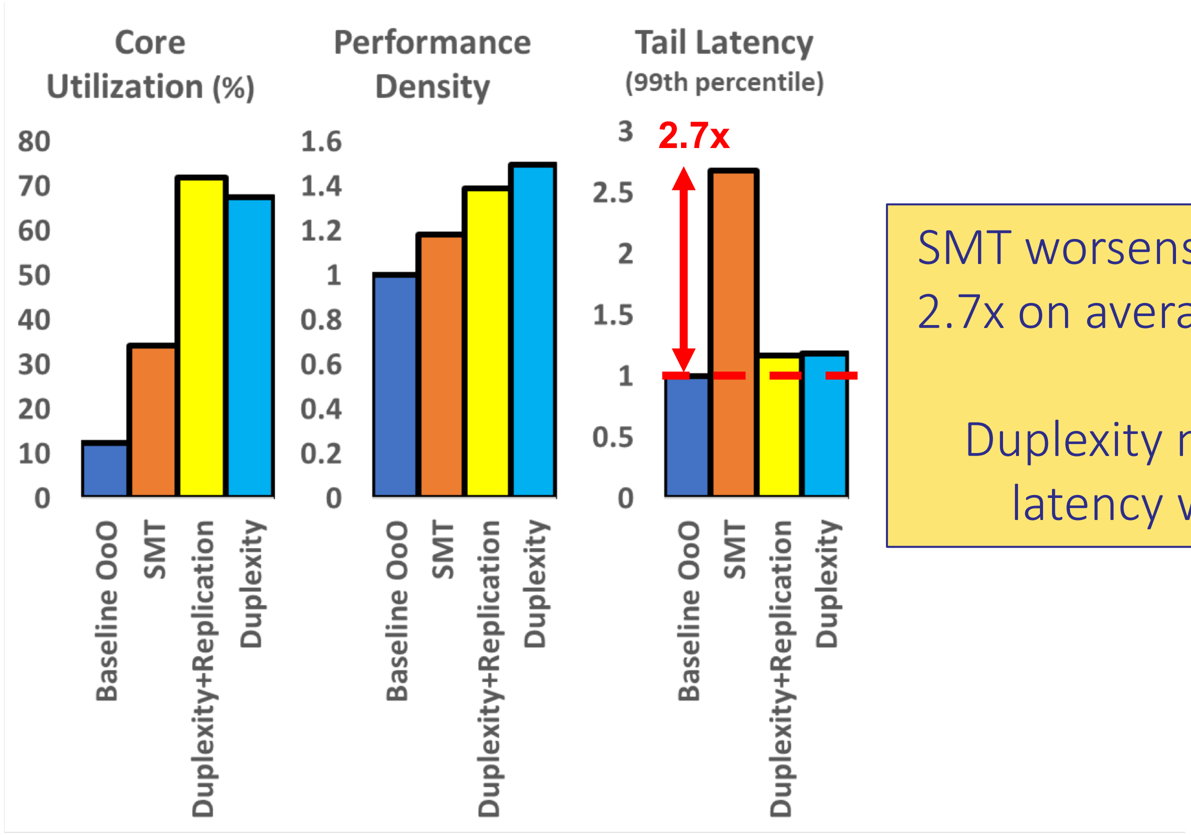
Duplexity achieves 34% higher average core utilization compared to SMT
Within 4% of the utilization achieved by Duplexity + replication

Evaluation



Duplexity improves performance density by 49%, 28%, and ***10%*** compared to baseline, SMT, and *Duplexity+Replication*

Evaluation



SMT worsens tail latency by 2.7x on average (up to 5.7x)

Duplexity maintains tail latency within 19%

Conclusions

- ***Killer Microseconds:*** Frequent µs-scale pauses in microservices
- Modern computing systems not effective in hiding microseconds

- Our proposal: ***Duplexity***
  - Cost-effective highly multithreaded server architecture
  - Heterogeneous design:
    - Master cores for latency-sensitive microservices
    - Lender cores for latency-insensitive batch application
  - Master core may “borrow” threads from the lender core to fill utilization holes
  - Cores protect their threads’ cache states to avoid QoS violations

Duplexity improves utilization by 4.8x while maintaining tail latency within 19%

Questions?