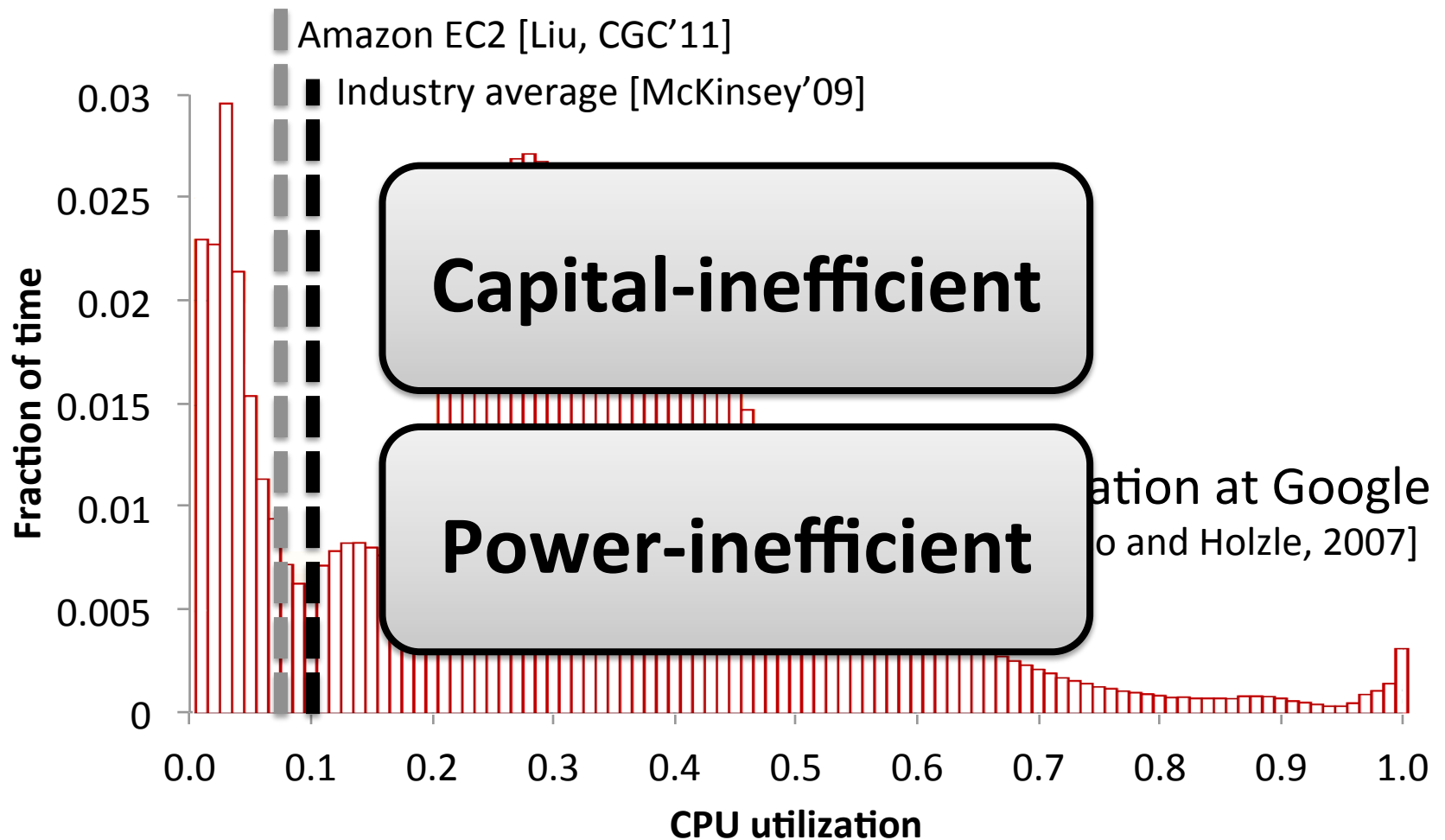


Reconciling High Server Utilization and Sub-millisecond Quality-of-Service

Jacob Leverich and Christos Kozyrakis,
Stanford University

EuroSys '14, April 14th, 2014

Server utilization is low



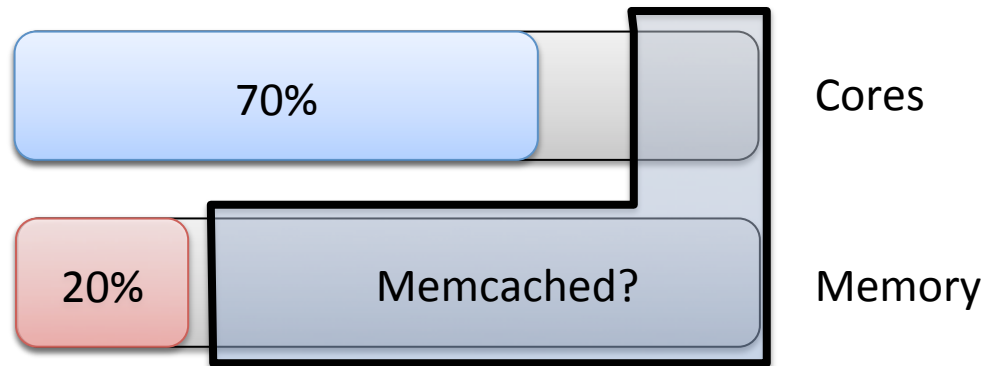
Why so low?

- Diurnal variation
- Capacity for future growth, unexpected spikes
- Server/workload mismatch

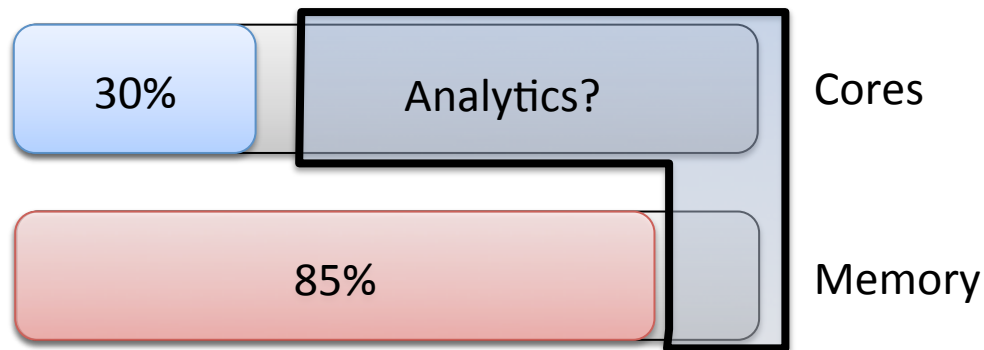
Simple solution:
Cluster Consolidation

Two consolidation examples

- Analytics cluster with unused memory



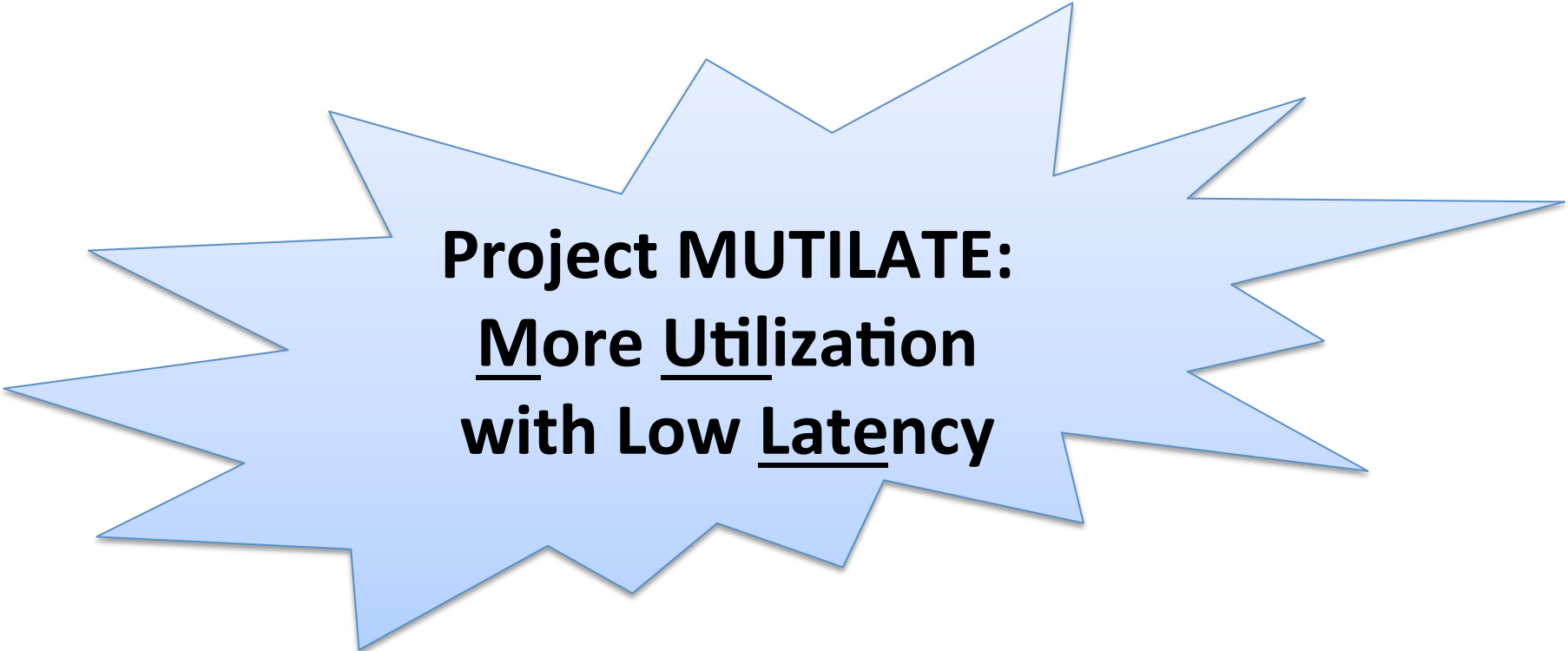
- Memcached cluster with unused CPU



Consolidation → Poor Performance & Quality of Service

- Interference on shared resources
 - Cores, caches, memory, storage, network
 - **QoS violations in low-latency applications**
 - Latency correlated with revenue [Mayer'06]
- Simple solutions lead to low-utilization
 - Don't co-locate work with low-latency services
 - Inflate reservations to reduce co-located jobs

Can we reconcile high utilization and
good quality of service?



**Project MUTILATE:
More Utilization
with Low Latency**

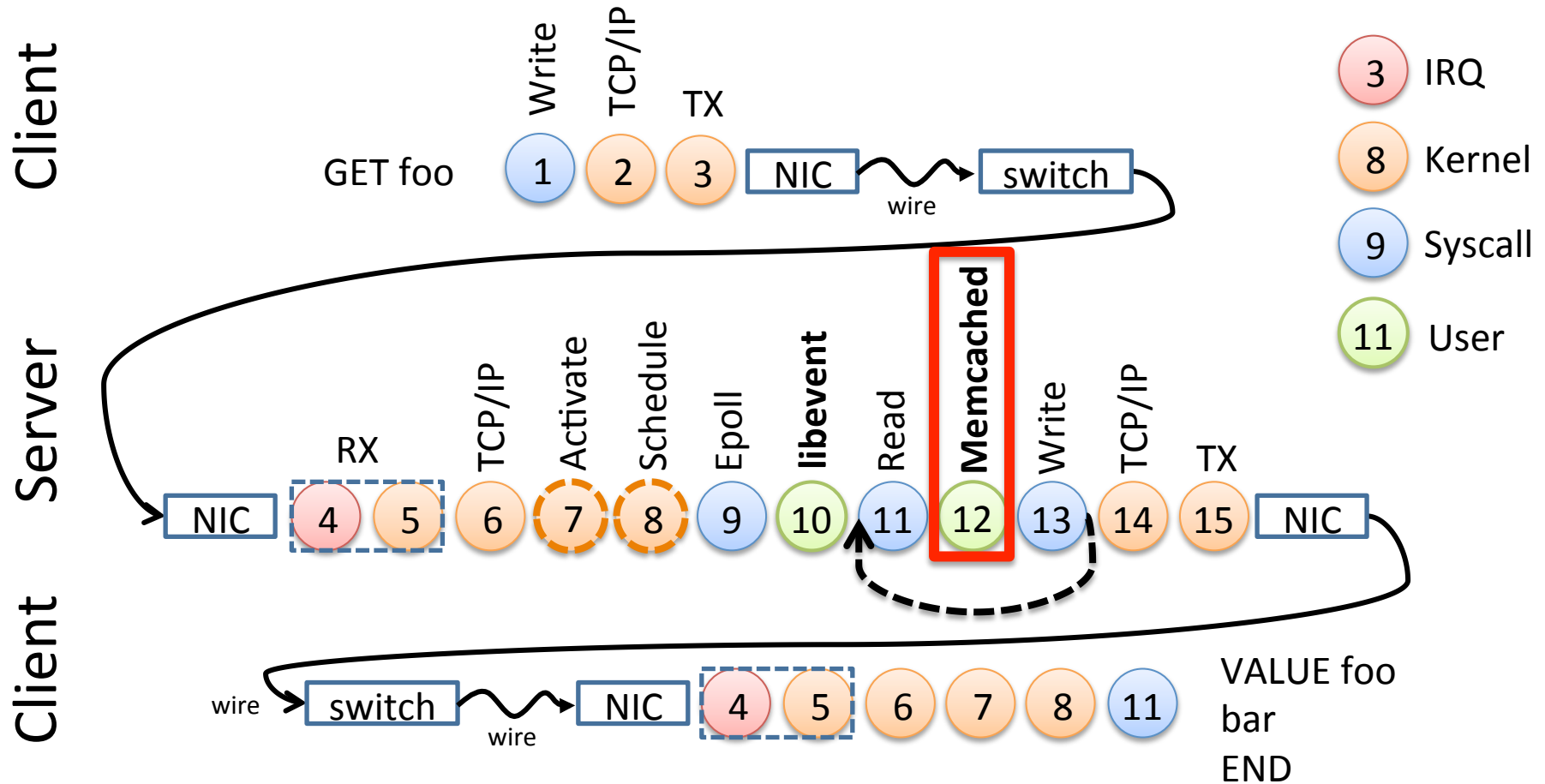
Contributions

- Identified key QoS vulnerabilities for sub-millisecond services
 - Queuing delay, scheduling delay, thread load imbalance
- Developed best practices to maintain good QoS
 - **Queuing delay:** Interference-aware provisioning
 - **Scheduling delay:** Use alternatives to CFS
 - **Thread load imbalance:** Dynamically share connections/requests [or pin threads]
 - **Network interference:** NIC receive-flow steering
- 17-52% reduction in TCO with good QoS despite interference

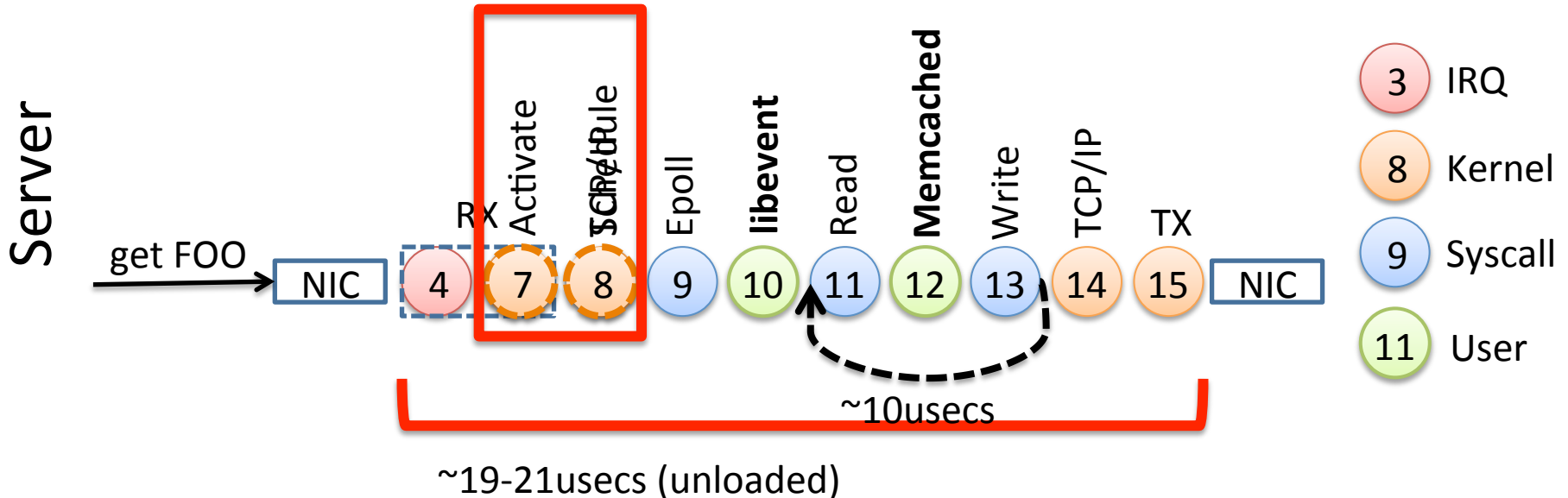
Focused on memcached

- Low nominal latency: 100s of usecs
- Sensitive to interference
- Good example of an event-based service
 - Arch. shared by REDIS, node.js, lighttpd, nginx, etc.
- Focus on interference due to consolidation
 - Ignore misbehaving clients, large requests, etc.
[Shue, OSDI'12, "Pisces"]

Life of a memcached request



QoS vulnerabilities

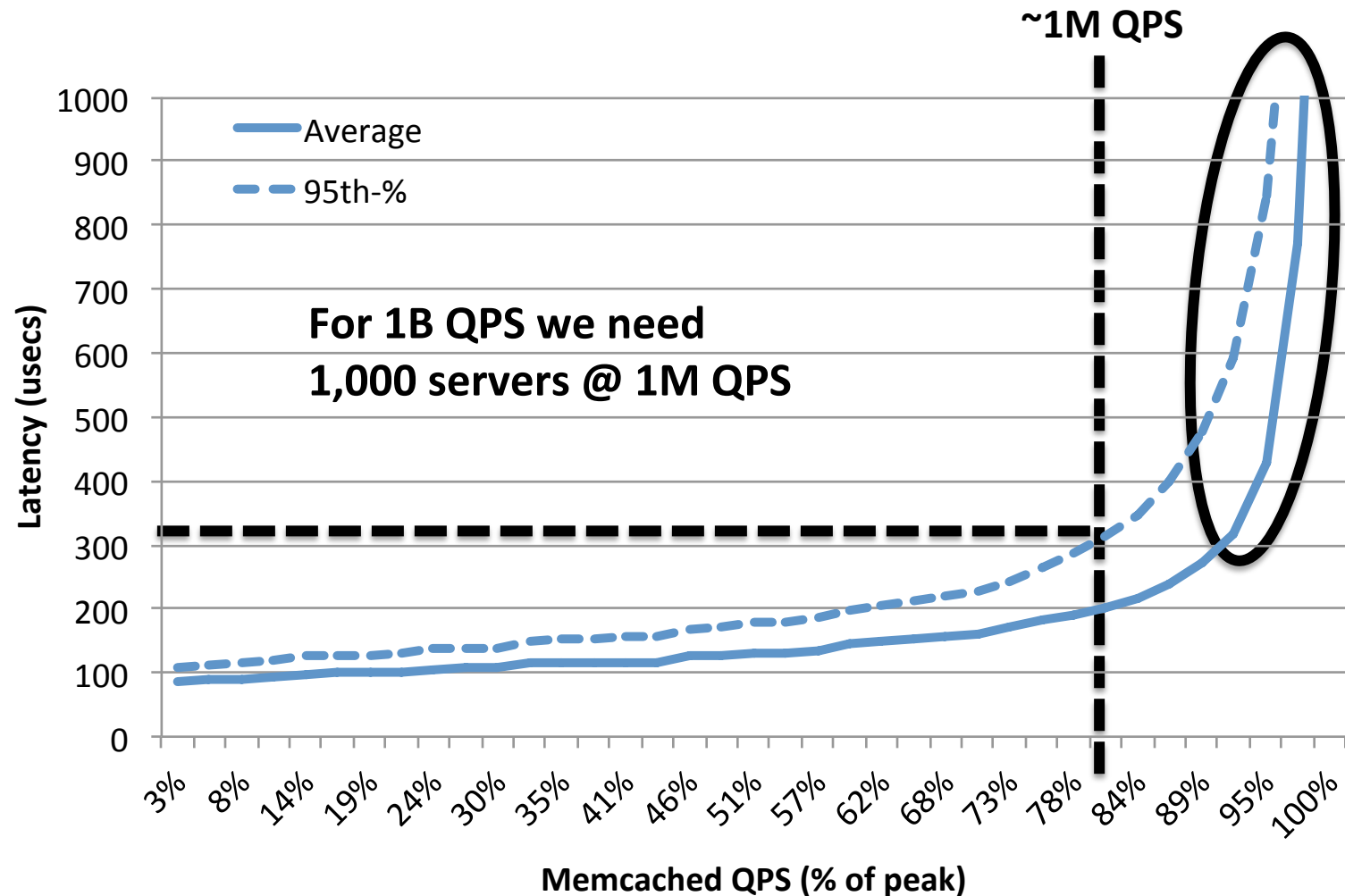


- Queuing delay
 - Function of load and service time
- Scheduling delay
 - Wait time and context switch latency

Let's capacity plan a cluster

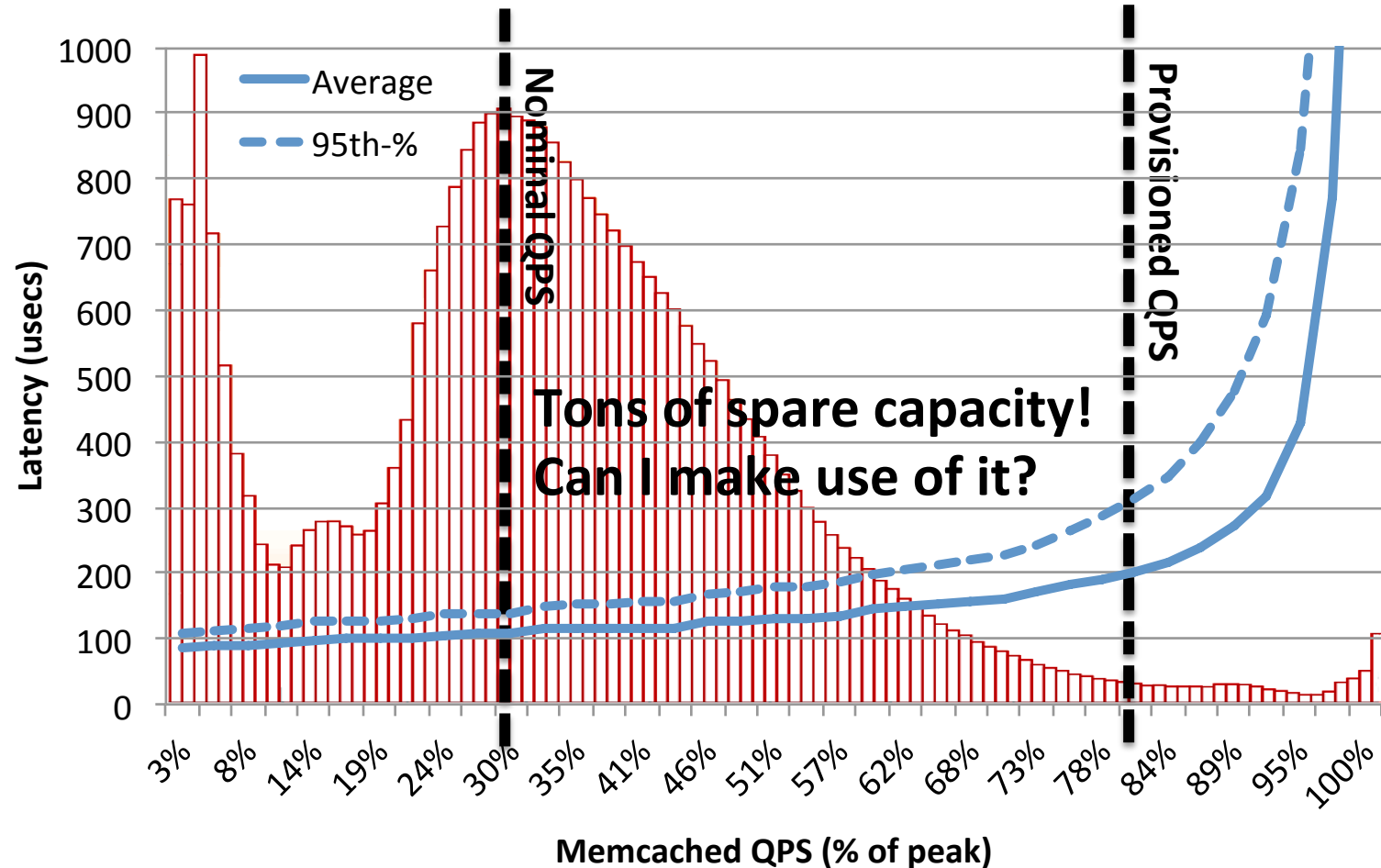
- Want to support 1B queries/sec total
 - Accounts for diurnal variation, unexpected spikes (worst-case peak)
 - Must maintain low latency
- How many servers do we need?

Provisioning for Quality of Service



Provisioning for Quality of Service

Histogram of CPU utilization @ Google [Barroso'07]

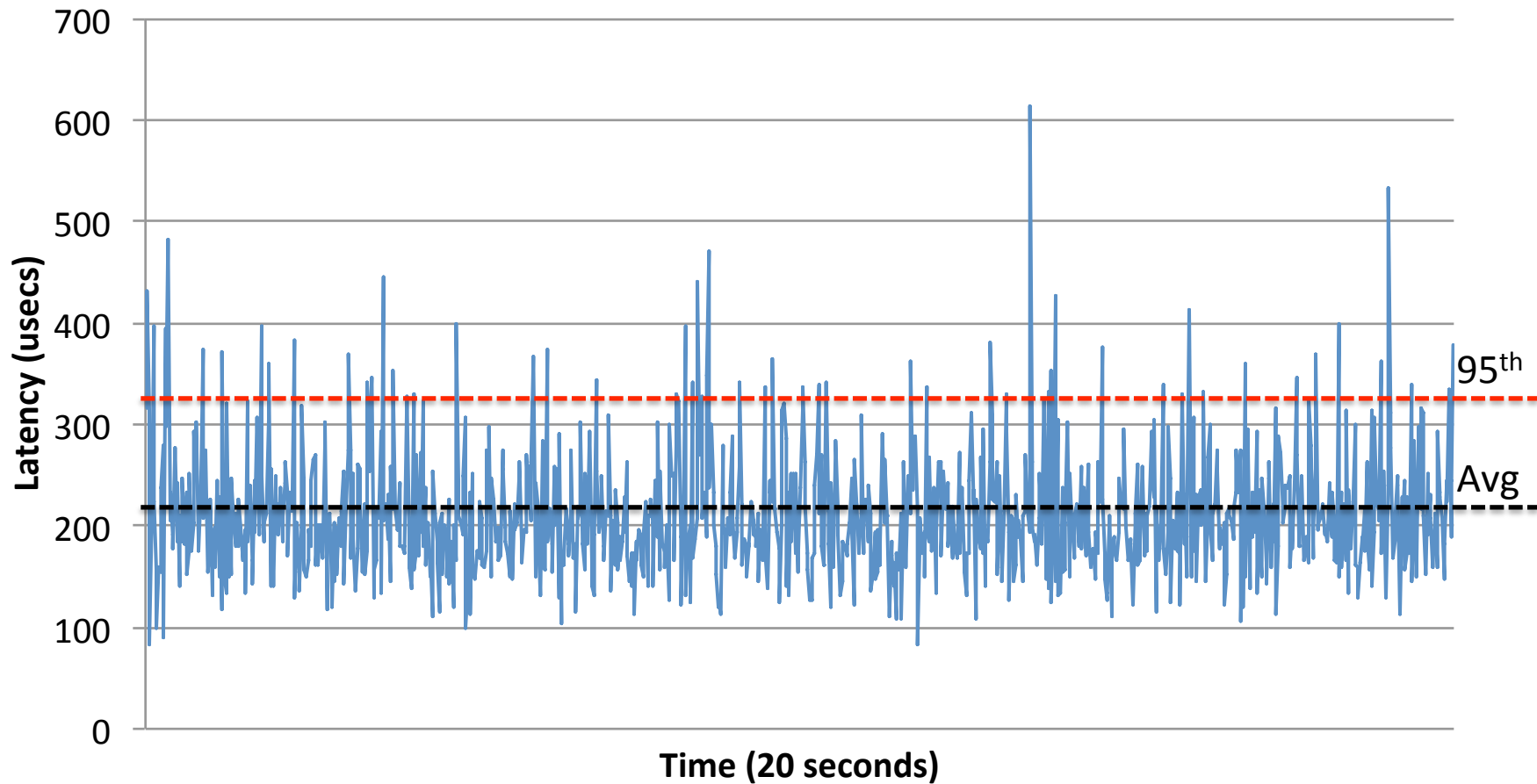


Cluster consolidation

- Memcached cluster
 - 1,000 servers, 30% nominal load
- Analytics cluster
 - 1,000 servers, 50% load, best-effort batch jobs
- Can we combine this capacity?
 - Must ensure we don't disturb provisioned QoS for the memcached server

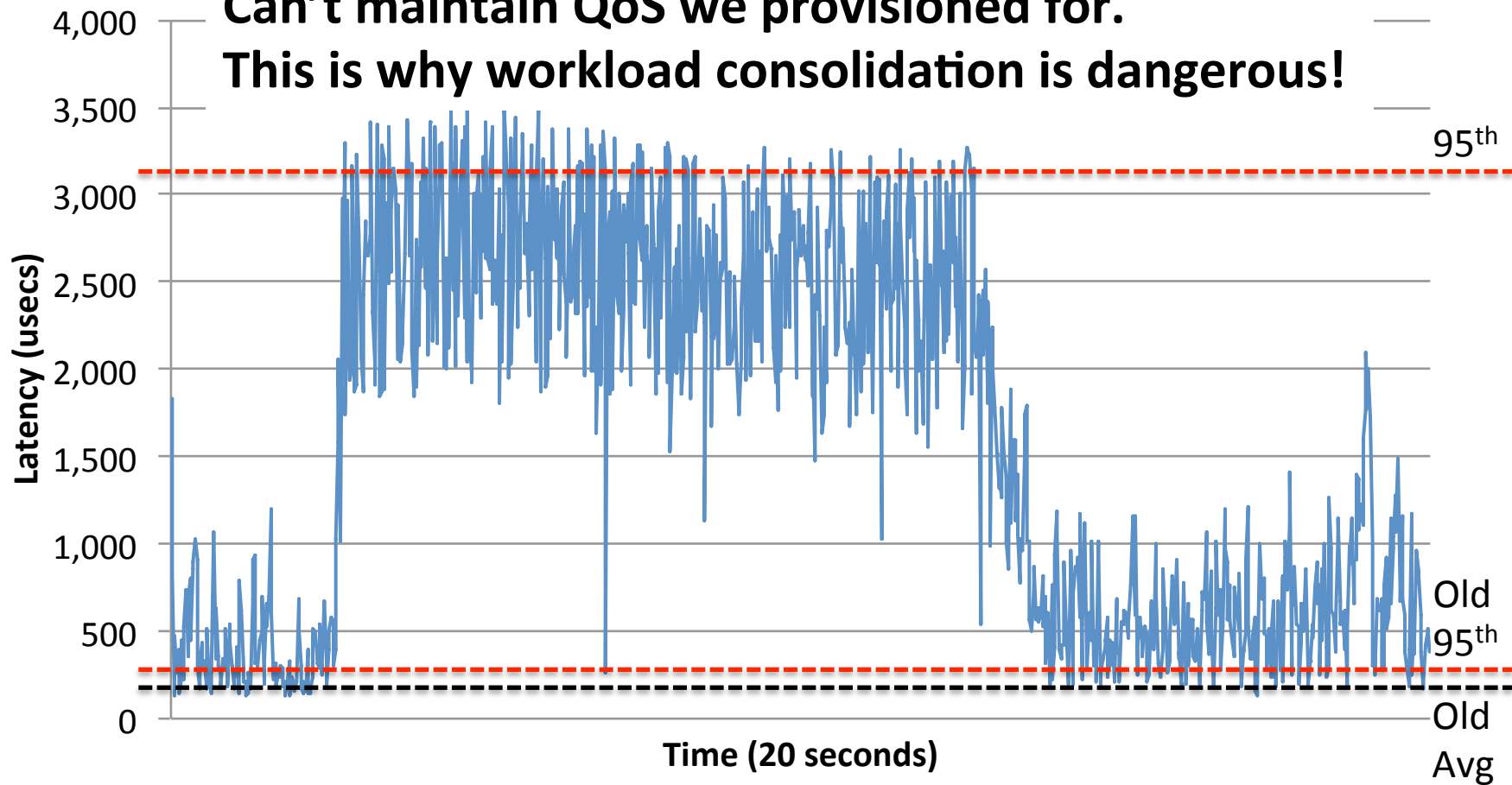
Latency @ 80% QPS

Baseline (no interference)



Latency @ 80% QPS with 471.omnetpp

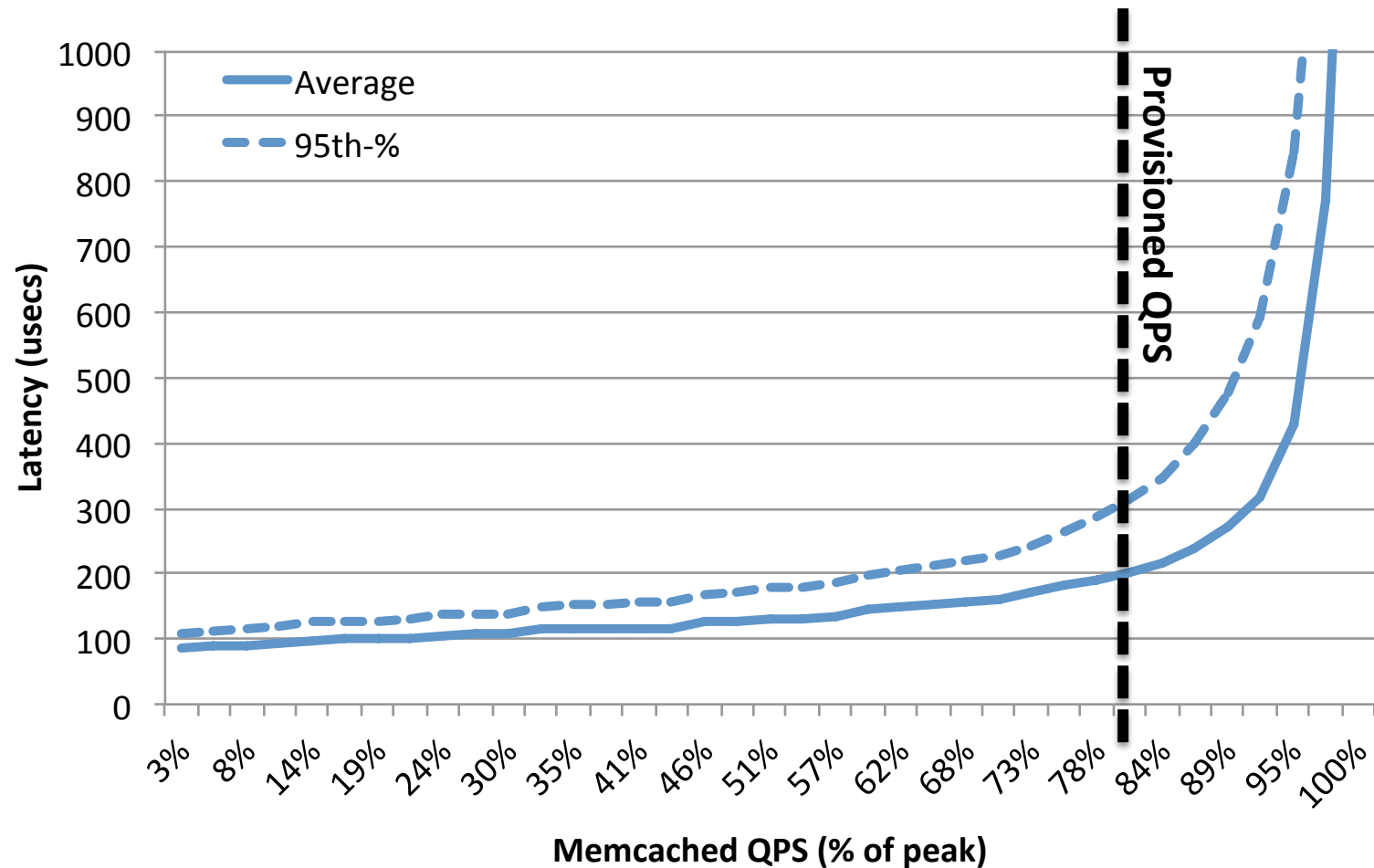
**Can't maintain QoS we provisioned for.
This is why workload consolidation is dangerous!**



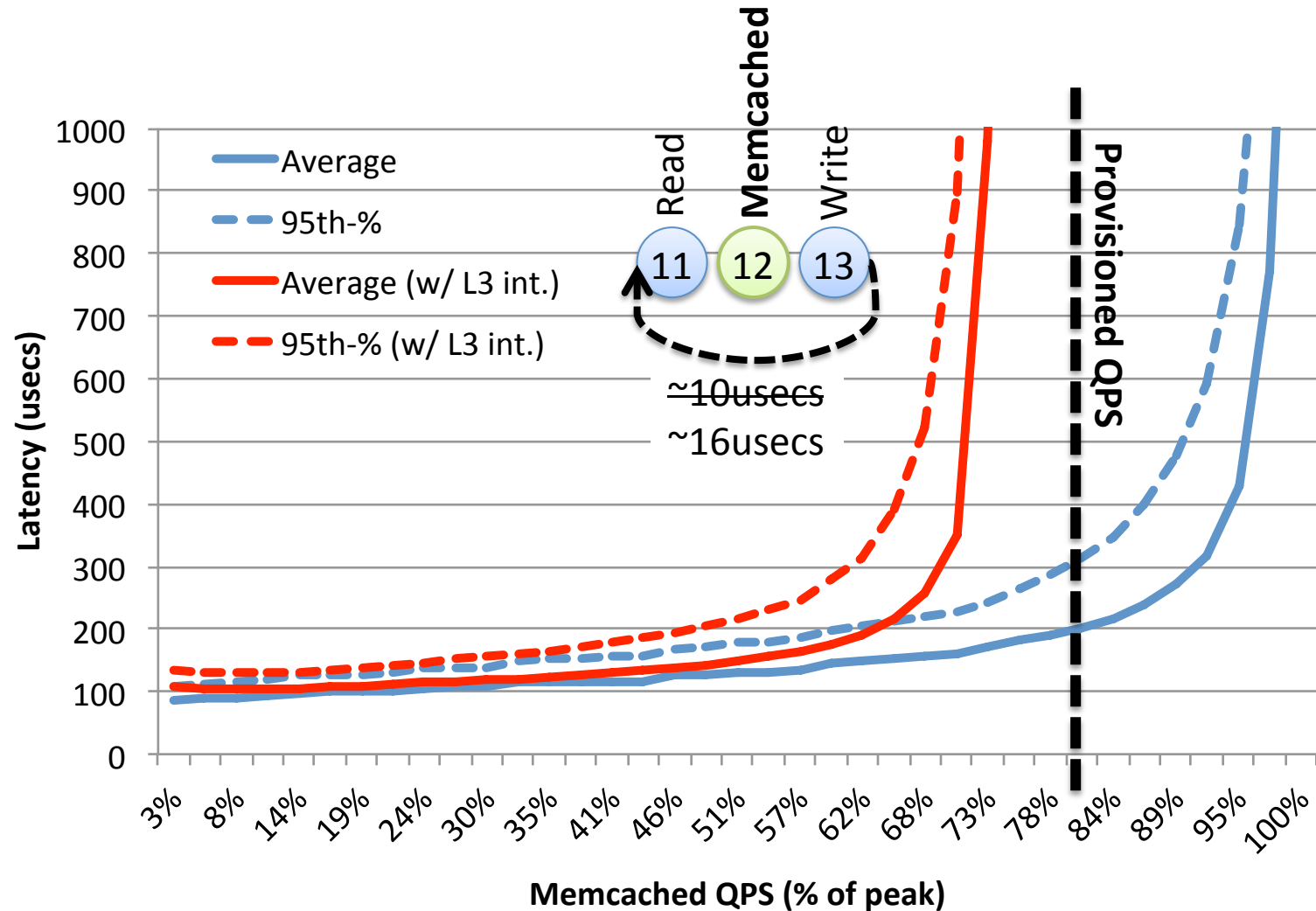
Related work

- CPI² [EuroSys'13]
 - Punish workload causing interference
- Bubble-Up, Paragon [MICRO'11, ASPLOS'13]
 - Identify or predict workloads that interfere, don't consolidate
- Manage symptoms, don't address causes

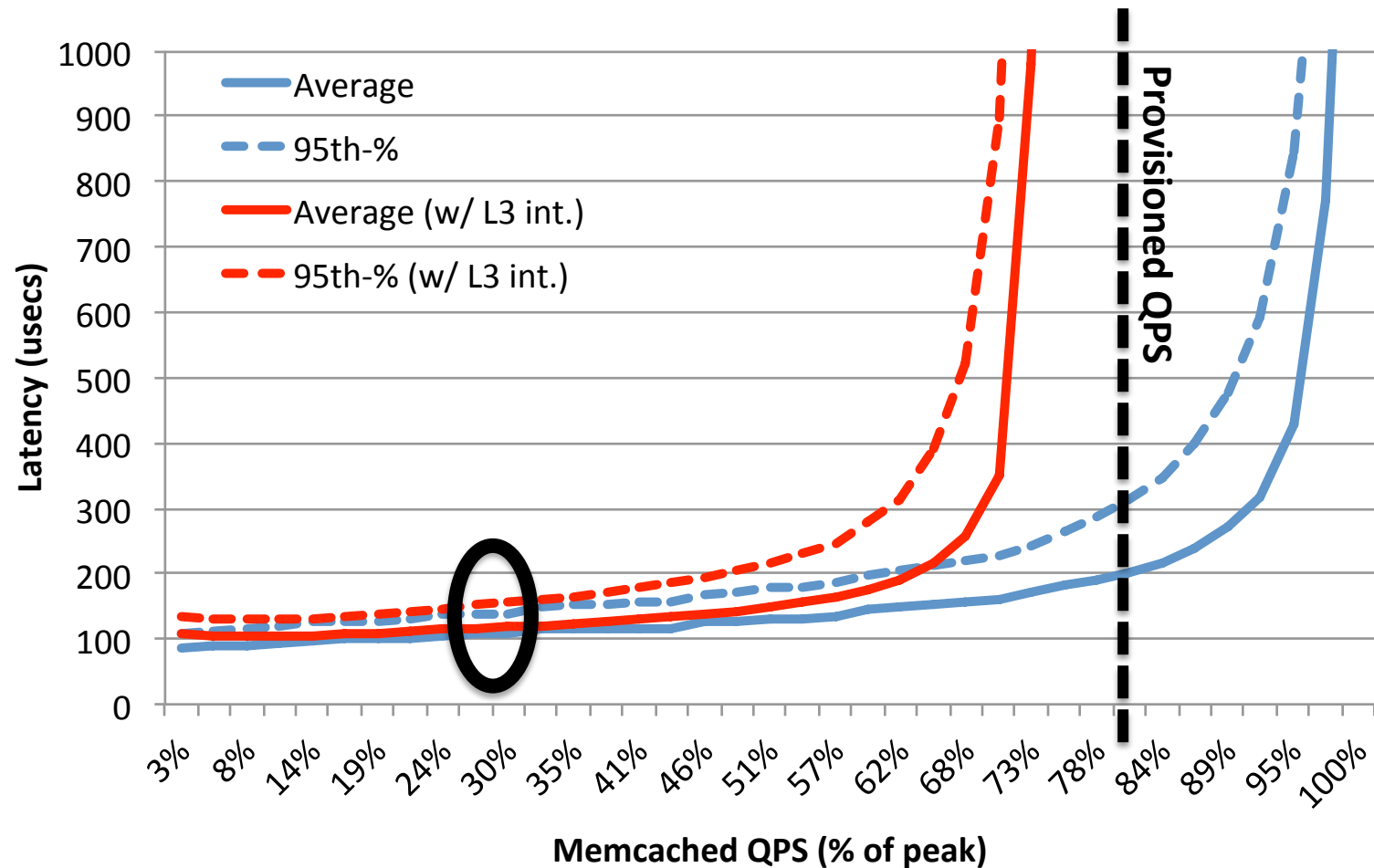
Latency with heavy L3 interference



Latency with heavy L3 interference



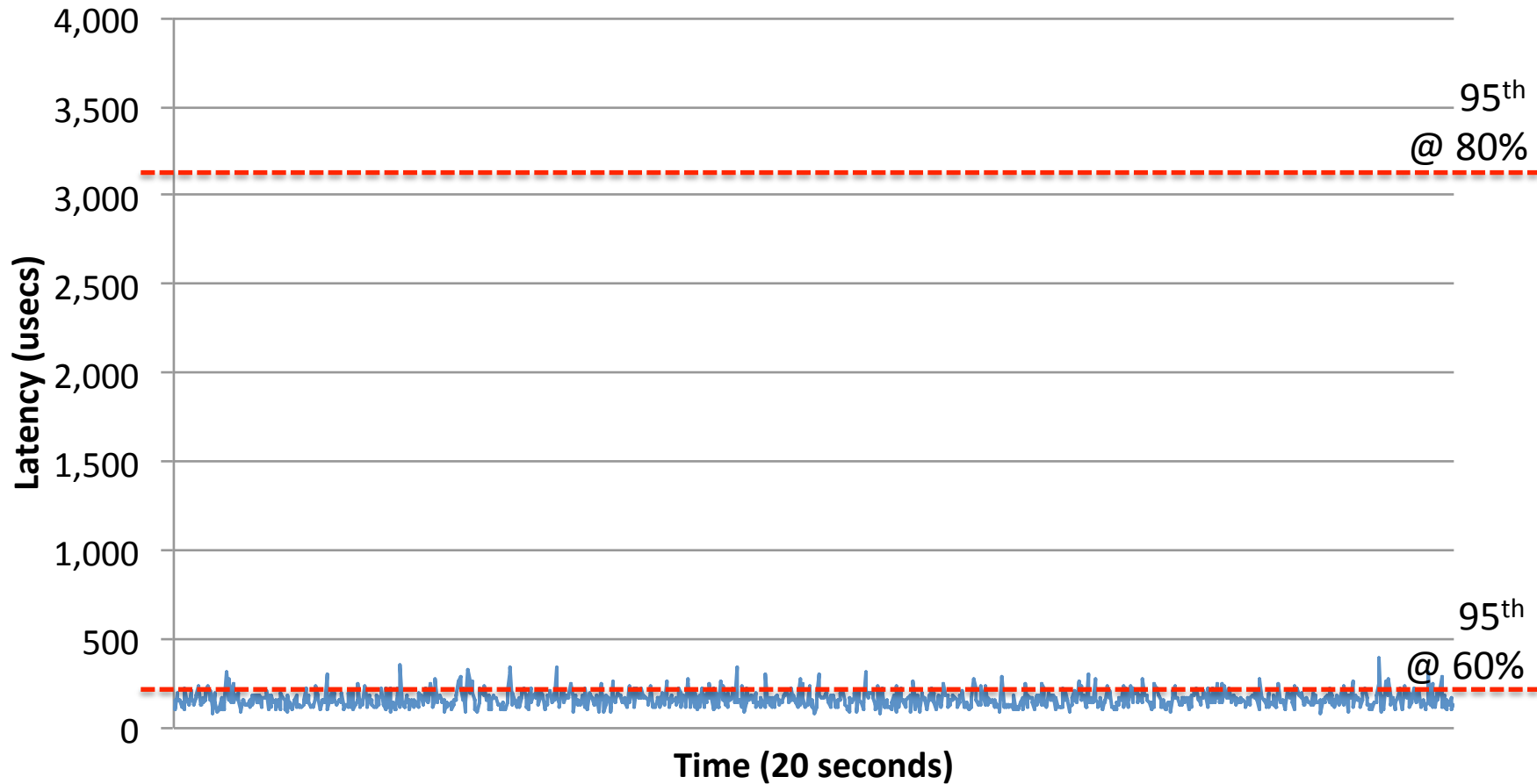
Latency with heavy L3 interference



Interference-aware provisioning

- Insight
 - Can live with interference, not queuing delay
 - Provisioning at 80% QPS gave us no margin of error
- Reprovision memcached cluster (60% QPS / server)
 - 1,300 servers for 1B QPS
 - Immune to cache interference, free to co-schedule jobs
 - Plenty of spare capacity for analytics cluster workload
(1,000 servers, 50% load)
- Combined 1,300 servers now at 60% nominal utilization
 - Achieve good QoS with interference, even at peak!
 - **33%** fewer servers overall,
23% less power consumption,
17% improvement in TCO

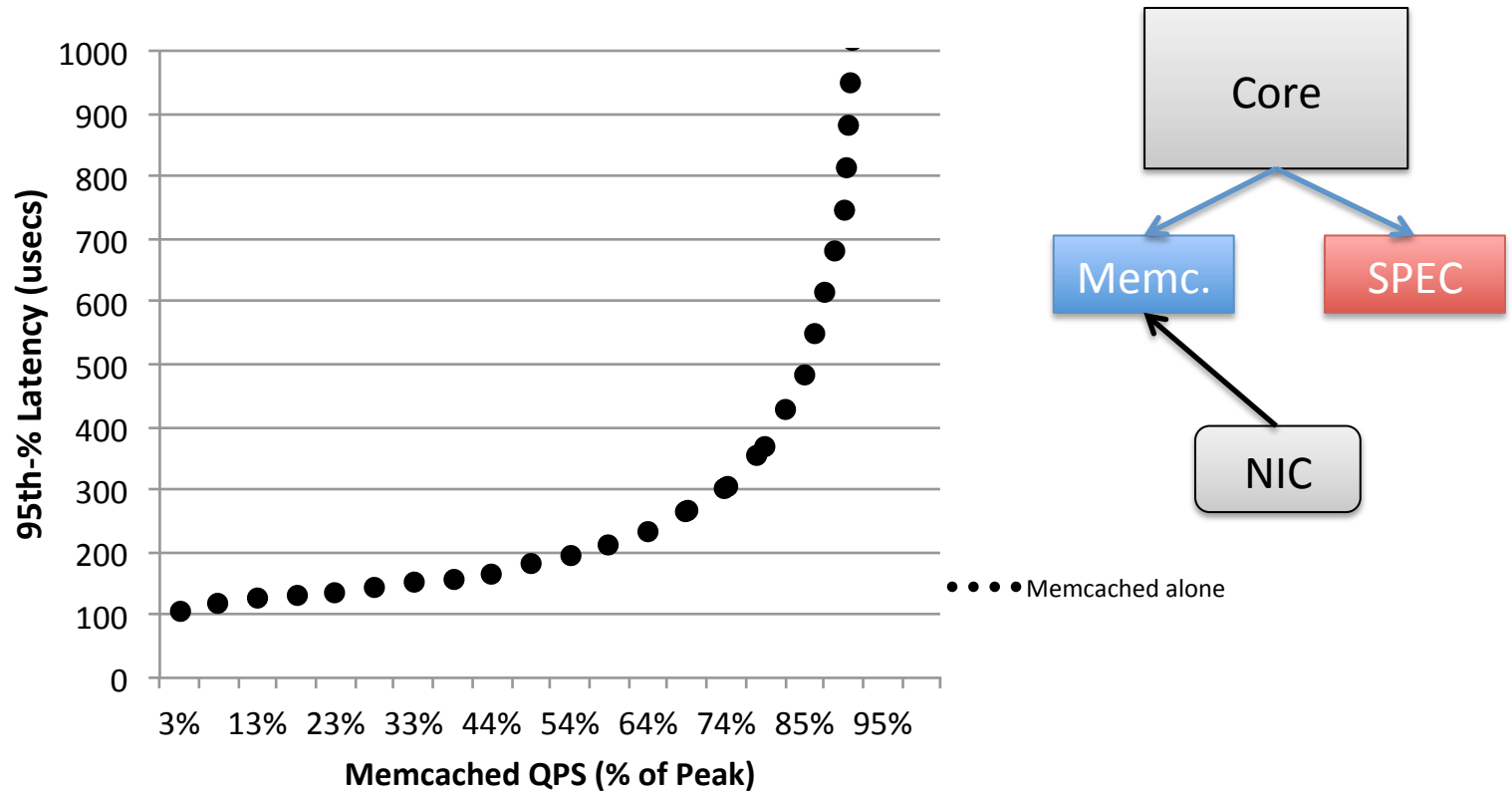
Latency @ 60% QPS with 471.omnetpp



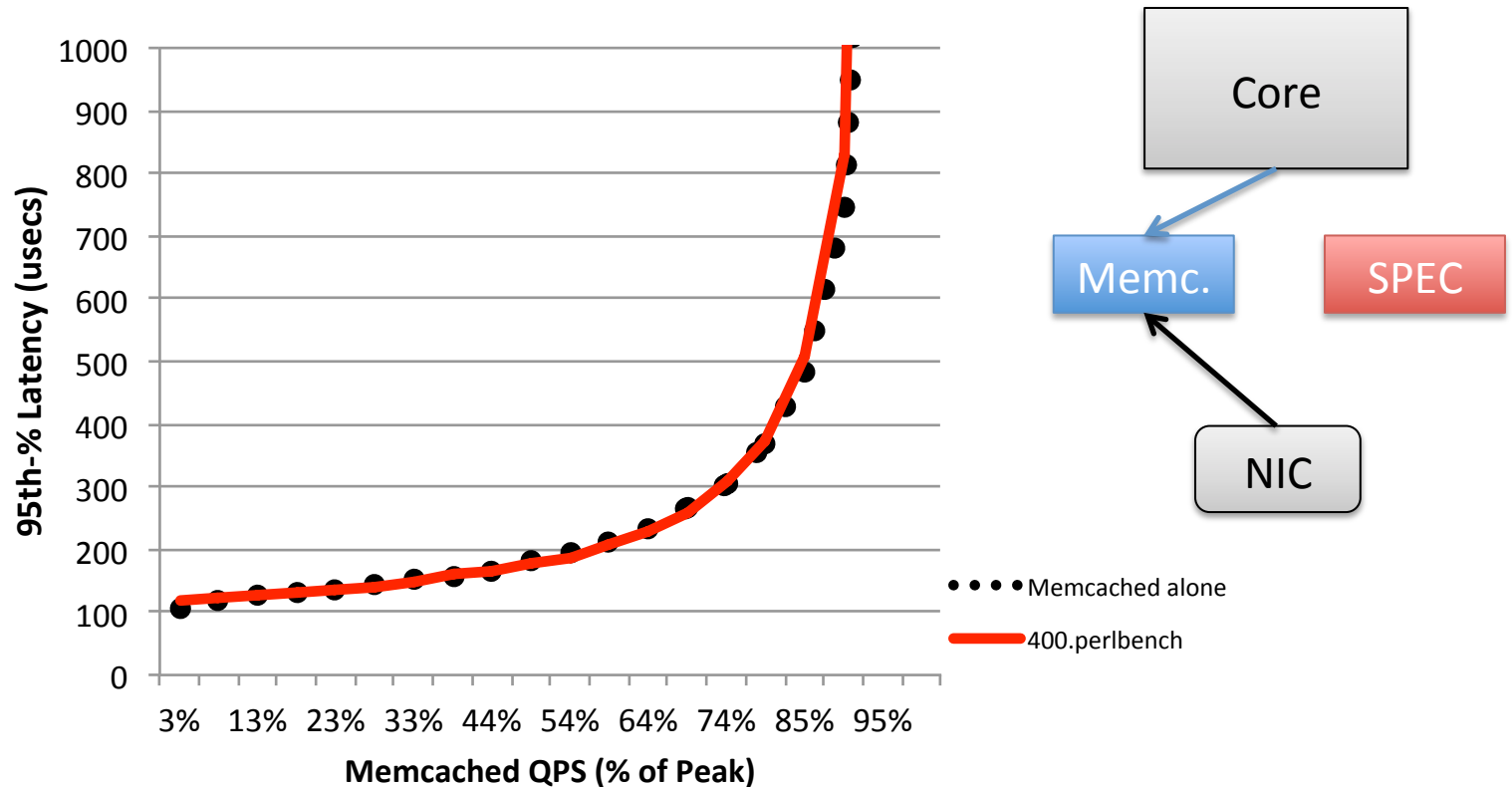
Scheduling delay

- Consolidated cluster (memcached + analytics)
 - Analytics = “best effort”, only use spare CPU time
 - Only worry about context switch latency

Time-sharing with memcached

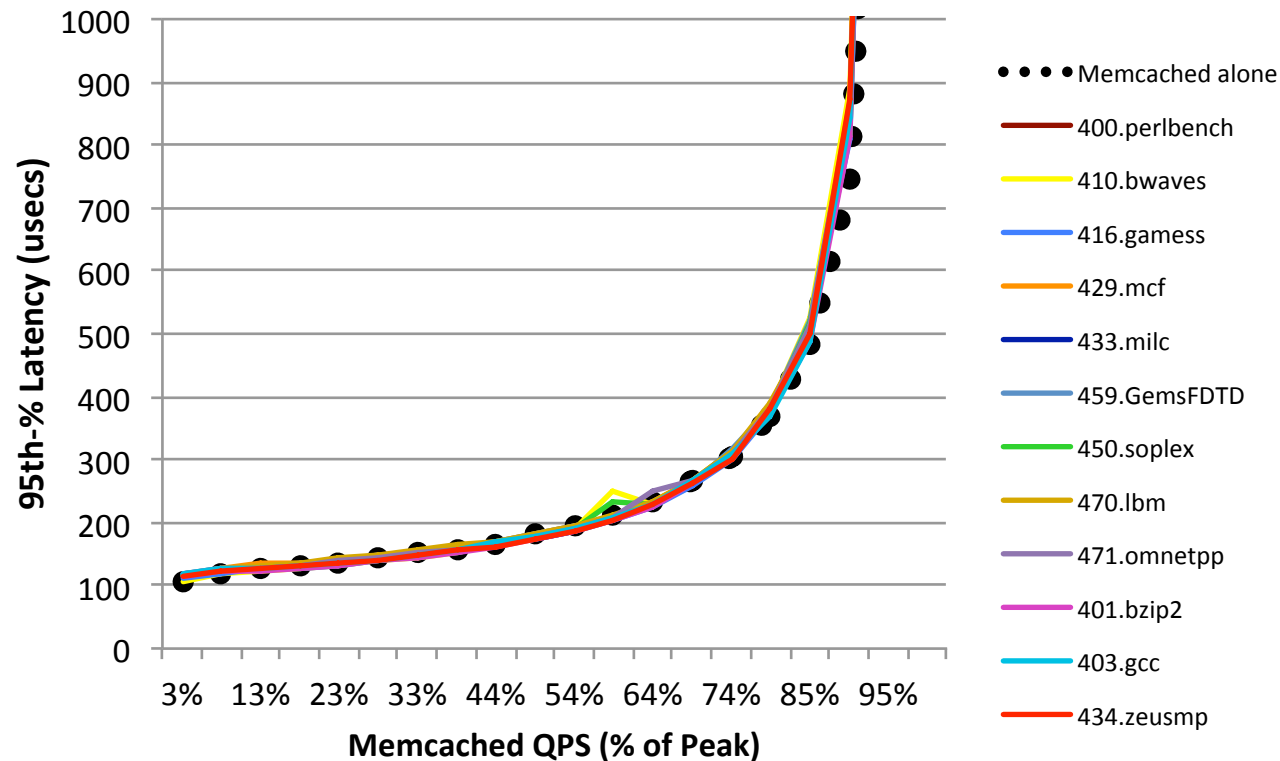


Time-sharing with memcached



- Context switch doesn't add any latency!

Time-sharing with memcached

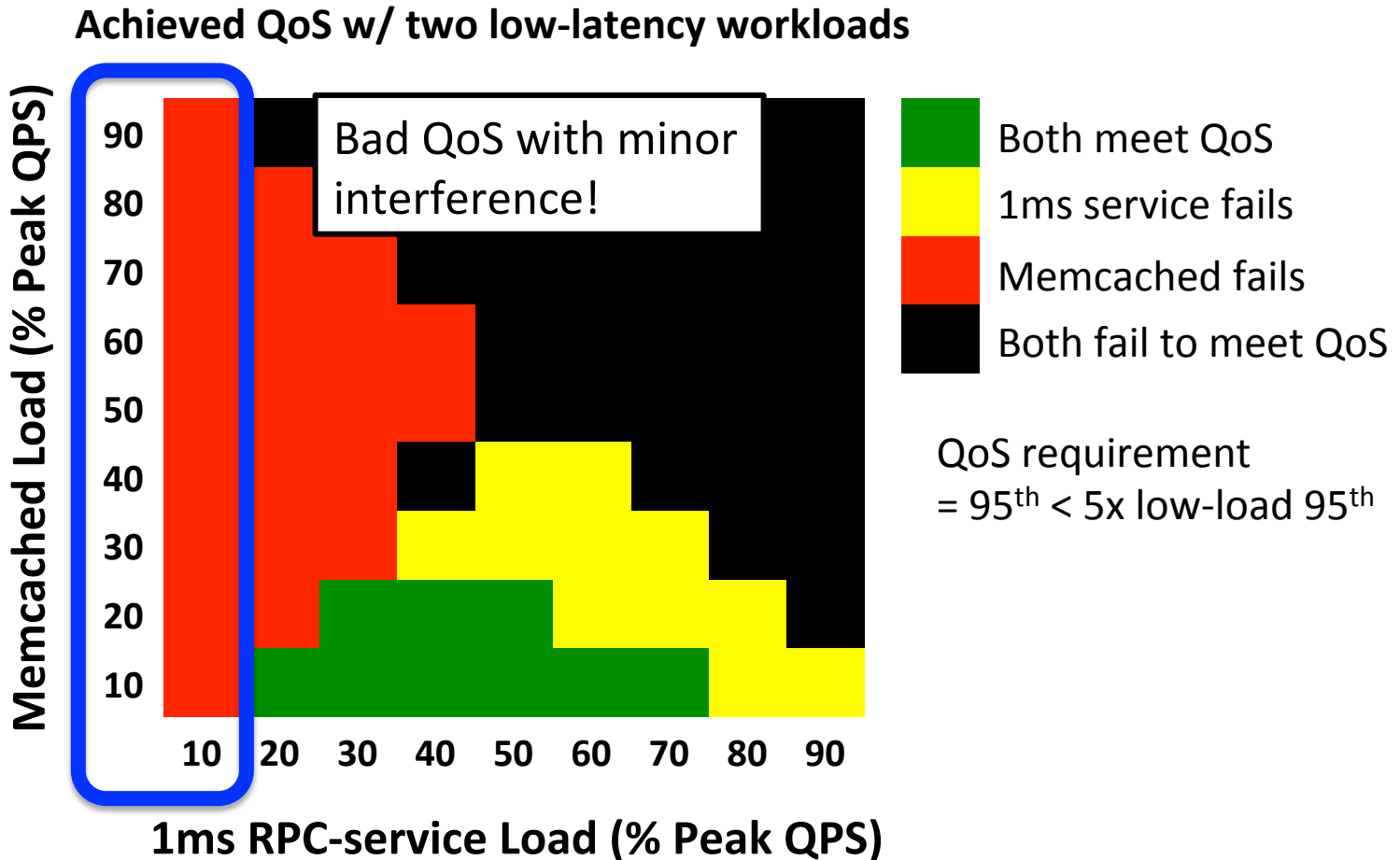


- Context switch doesn't add any latency
- Insensitive to workload

Scheduling delay

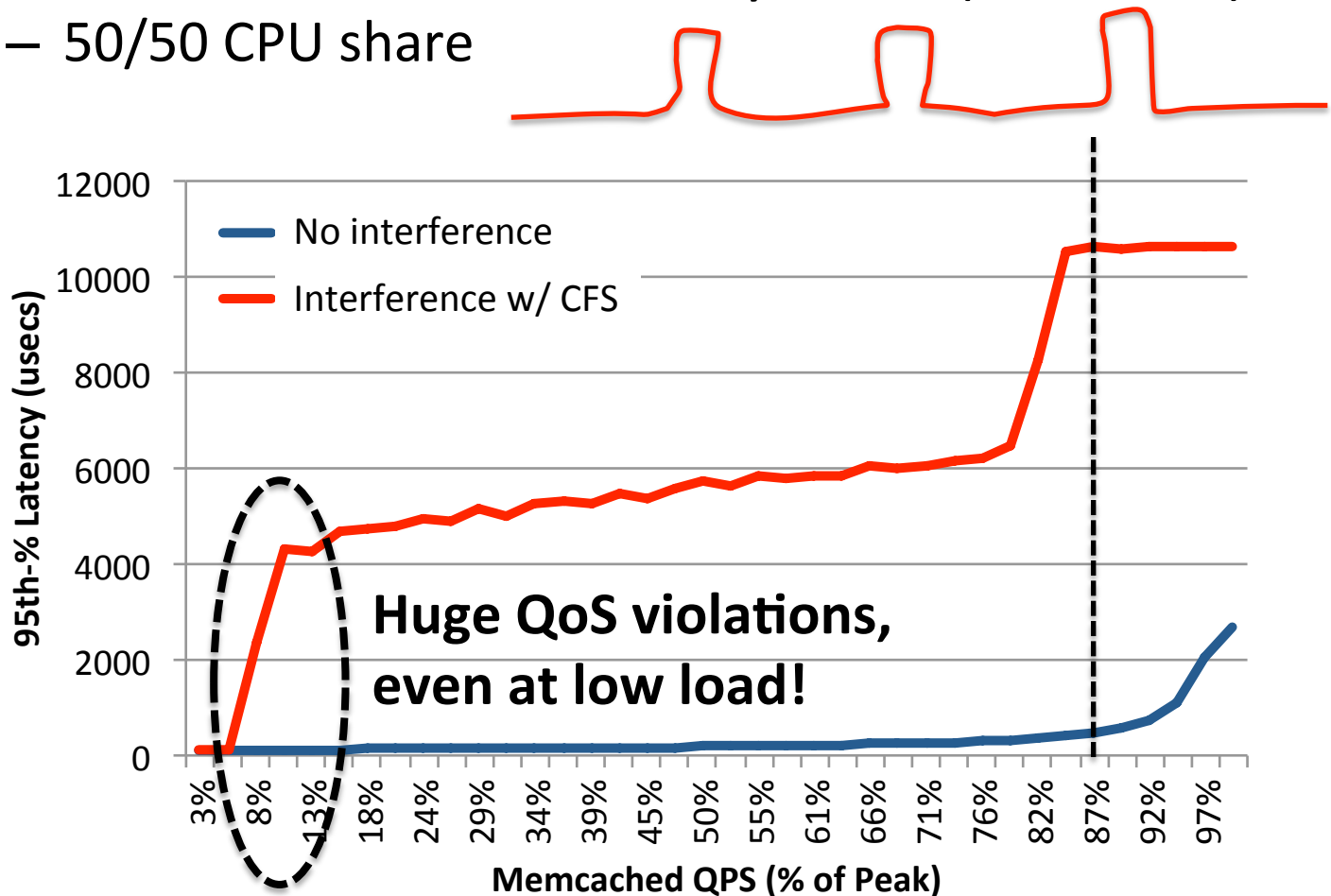
- Consolidated cluster (memcached + analytics)
 - Analytics = “best effort”, only use spare CPU time
 - Only worry about context switch latency
 - No worries!
- Co-scheduling low-latency applications
 - i.e. memcached + 1ms RPC service

CFS can't guarantee low latency



Time-sharing with a periodic task

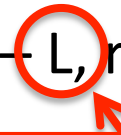
- Periodic task runs 6ms every 48ms (12% load)
 - 50/50 CPU share



CFS: Completely Fair Scheduler

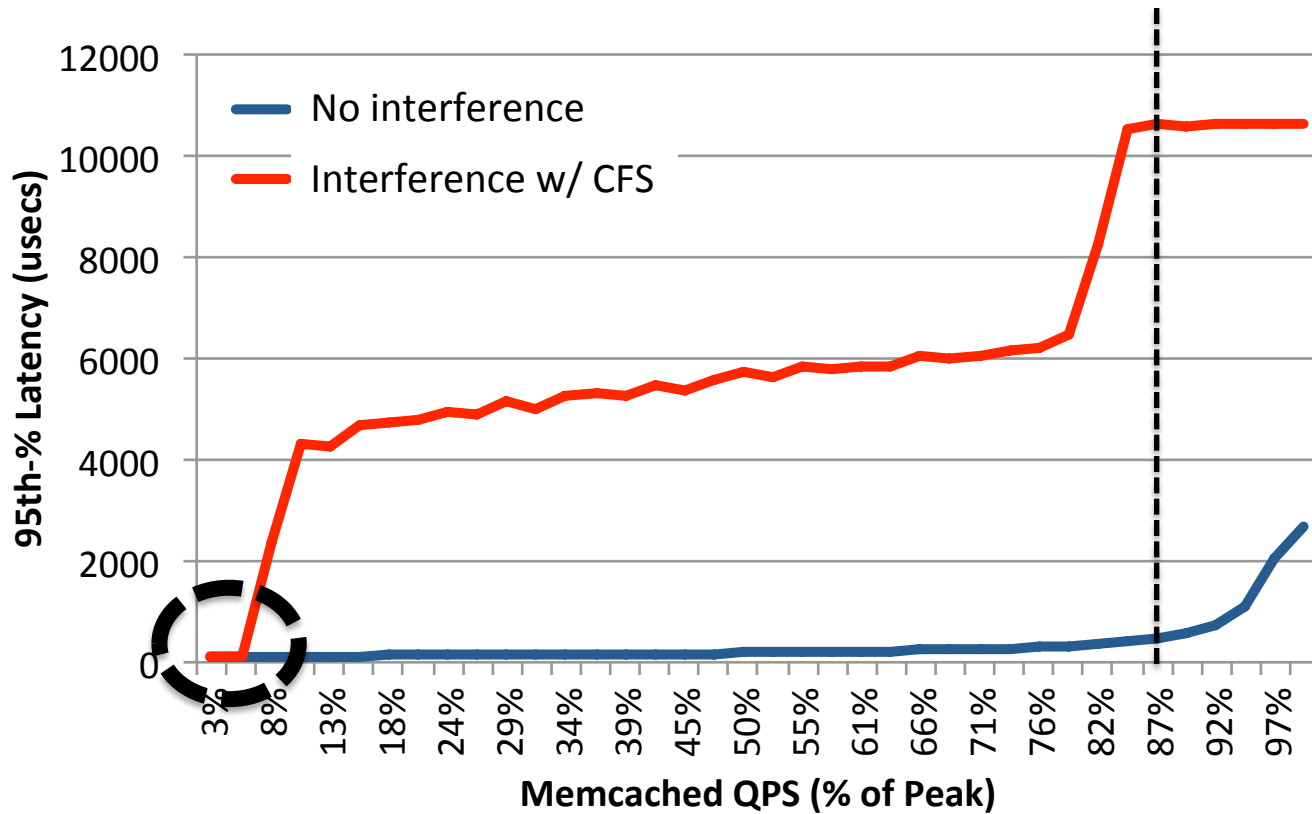
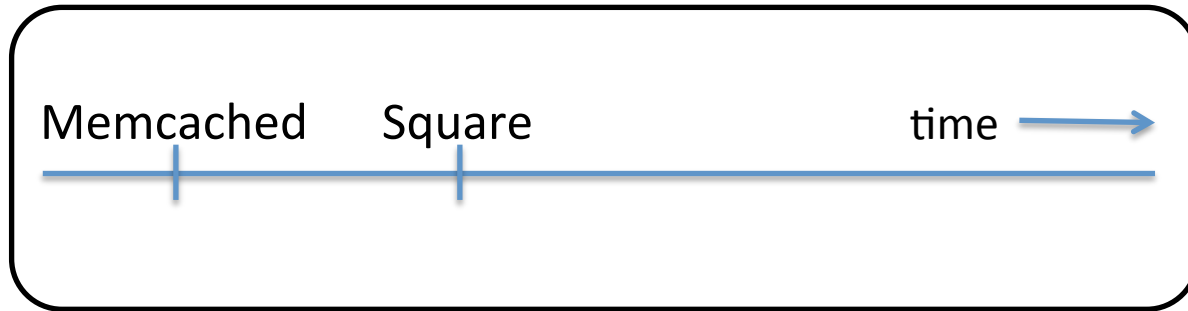
- Tasks sorted by runtime along a timeline
 - Run the earliest task whenever you reschedule
 - When a task T wakes up, assign its runtime as:

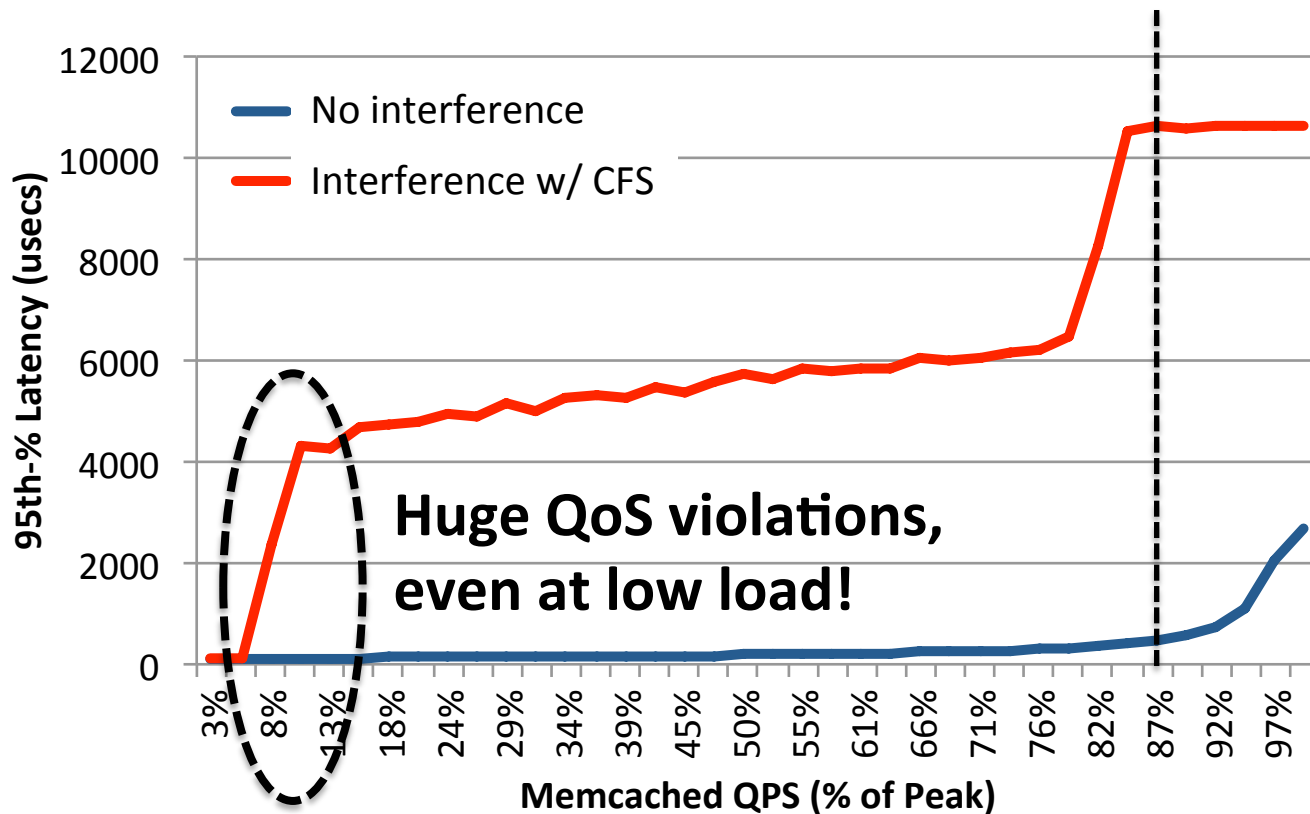
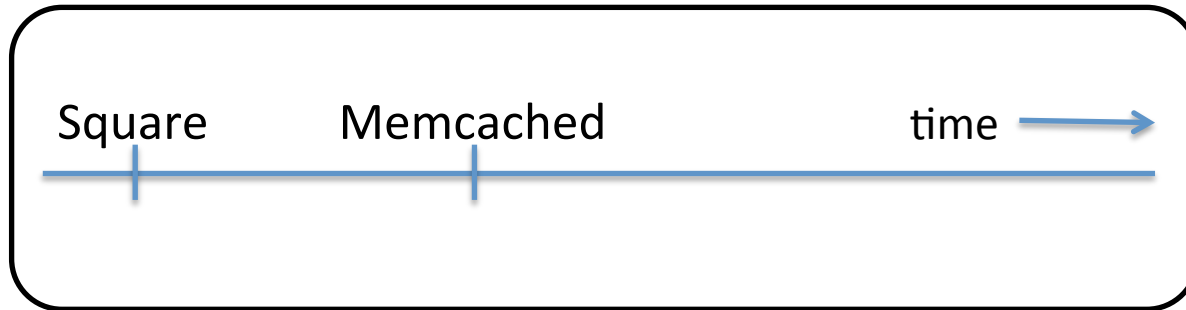
$$\text{runtime}(T) = \max(\min(\text{runtime}(*)) - L, \text{runtime}(T))$$



s by default!

Good for desktop
Bad for datacenters!

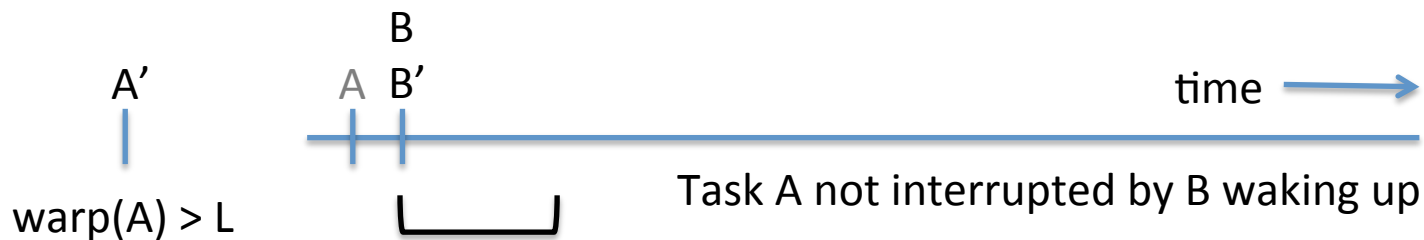




BVT: Borrowed virtual time [Duda'99]

- Same basic principle as CFS, plus
 - Assign each task a “warp” value
 - Schedule based on “effective” runtime:

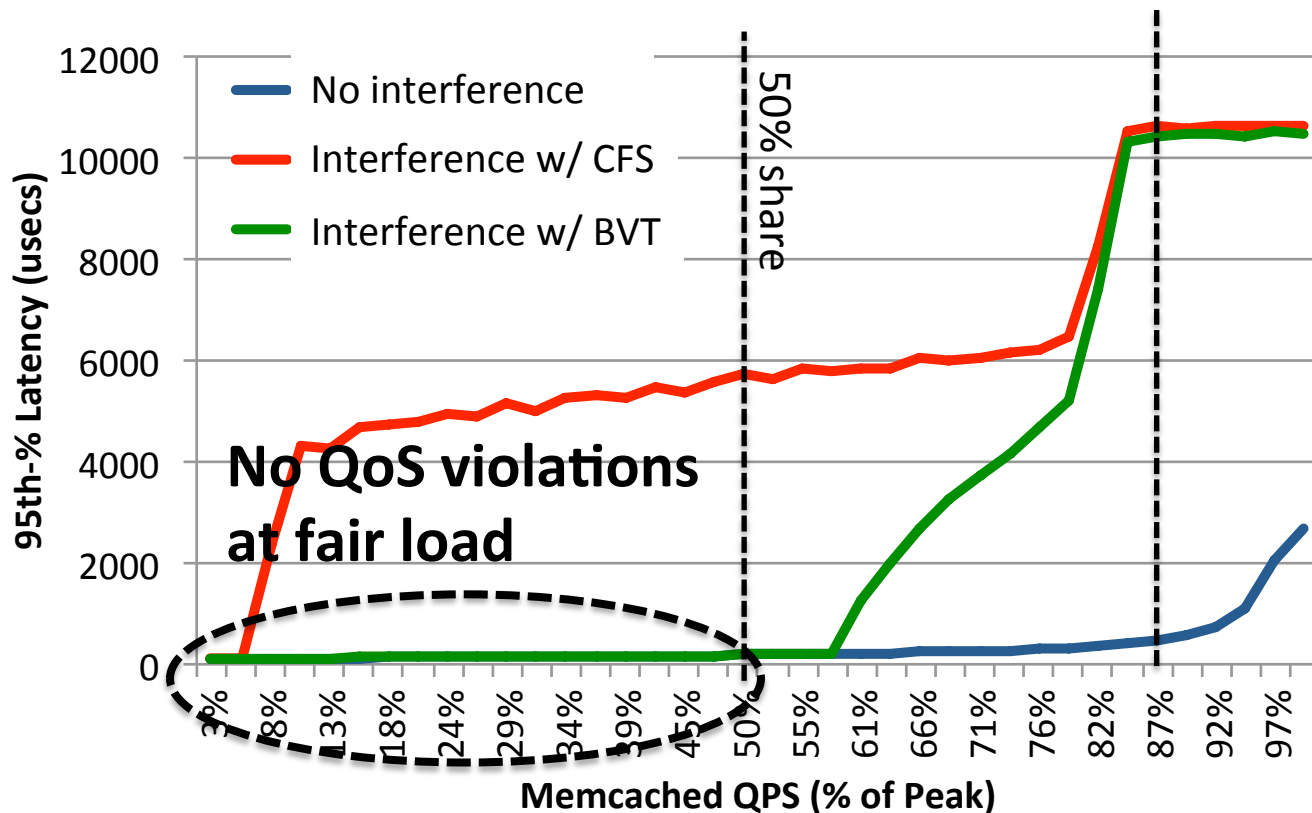
$$\text{effective runtime}(T) = \text{runtime}(T) - \text{warp}(T)$$



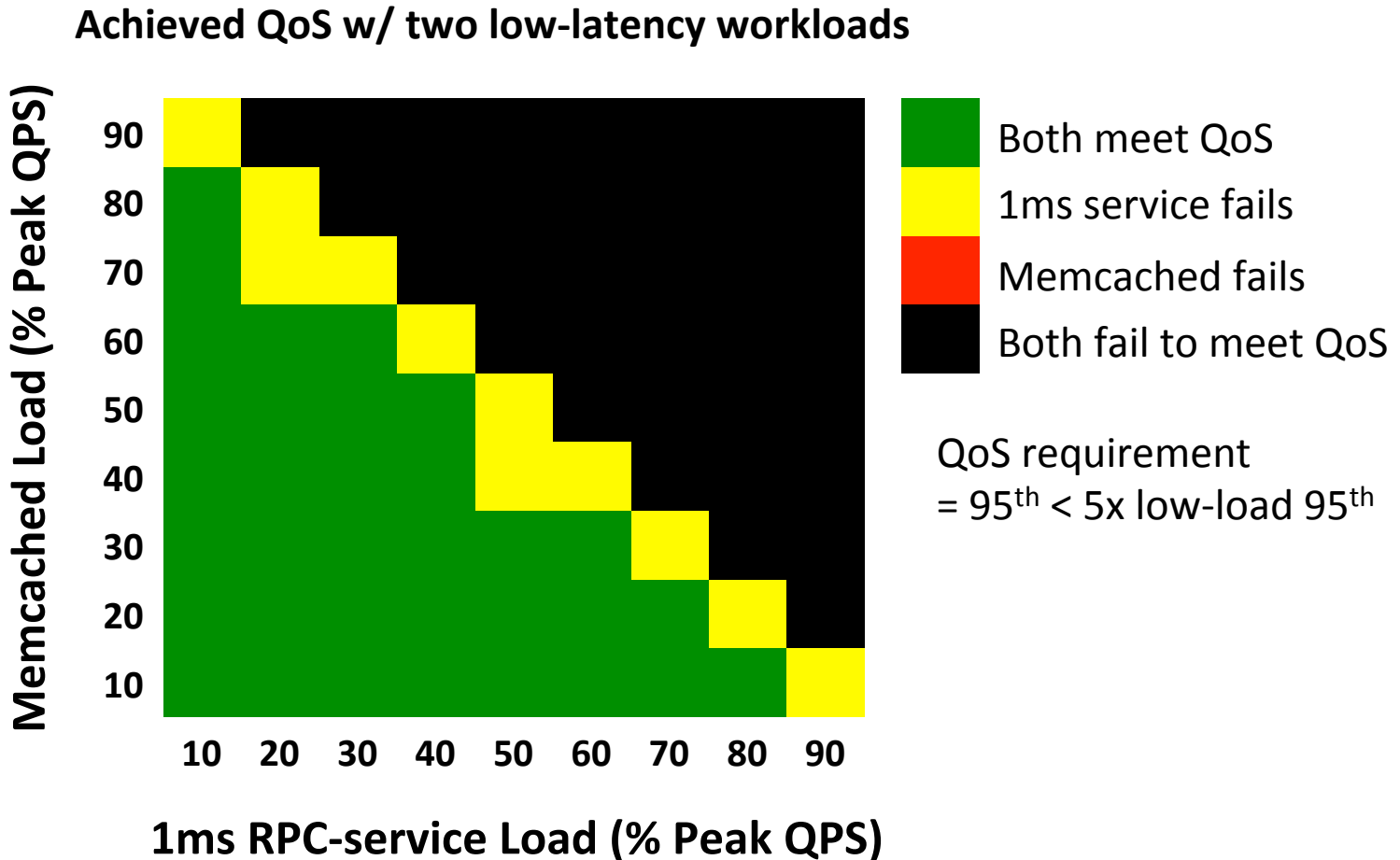
- Short-term ~~term~~ preemption bias
- Long-term throughput fairness

BVT: Borrowed virtual time [Duda'99]

- Periodic task runs 6ms every 48ms (12% load)
 - 50/50 CPU share



BVT: Good QoS for both services



BVT patch to Linux 3.5.0

- Simple extension to CFS
- Implemented at container group level
 - Warp specified per cgroup in cgroupfs
 - Defaults to normal CFS behavior
- <https://gist.github.com/leverich/5913713>

```
include/linux/sched.h | 8 ++++++
init/Kconfig          | 14 ++++++++
kernel/sched/core.c   | 31 ++++++++
kernel/sched/fair.c    | 93 ++++++++
+++++-----
kernel/sched/features.h | 2 ++
kernel/sched/sched.h   | 4 +++
kernel/sysctl.c        | 9 ++++++
7 files changed, 158 insertions(+), 3 deletions(-)
```

Contributions

- Identified key QoS vulnerabilities for sub-millisecond services
 - Queuing delay, scheduling delay, thread load imbalance
- Developed best practices to maintain good QoS
 - Queuing delay: Interference-aware provisioning
 - Scheduling delay: Use alternatives to CFS
 - Thread load imbalance: Dynamically share connections/requests [or pin threads]
 - Network interference: NIC receive-flow steering
- Question: “Can we reconcile high utilization and good quality of service?”

Contributions

- Identified key QoS vulnerabilities for sub-millisecond services
 - Queuing delay, scheduling delay, thread load imbalance
- Developed best practices to maintain good QoS
 - Queuing delay: Interference-aware provisioning
 - Scheduling delay: Use alternatives to CFS
 - Thread load imbalance: Dynamically share connections/requests [or pin threads]
 - Network interference: NIC receive-flow steering
- Question: “Can we reconcile high utilization and good quality of service?”

Thanks!

Backup Slides

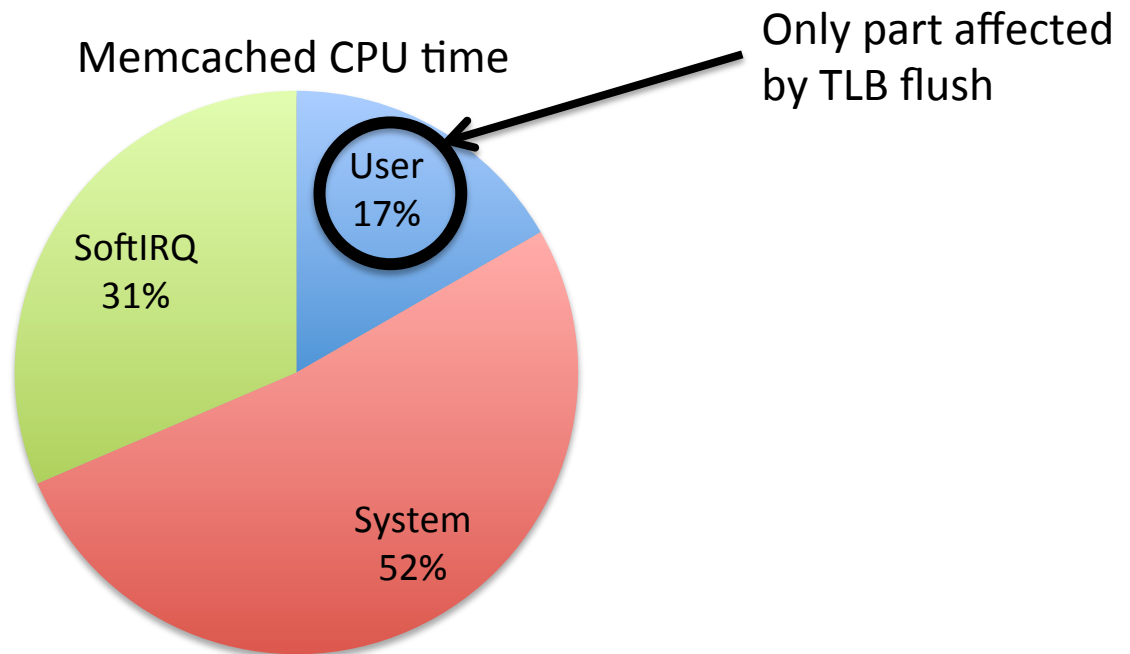
Mutilate:

A memcached load generator

- <http://github.com/leverich/mutilate>
- Distributed and epoll-based
- High performance (millions of QPS)
- Arbitrary intertransmission dist. (QPS control)
- Latency sampled at 1 kHz by independent open-loop connections (no client-side queuing delay due to load generation), UDP or TCP
- Arbitrary value/key size distributions (can replay “Workload Analysis of a Large-Scale Key-Value Store” [Atikoglu et al., SIGMETRICS’12])

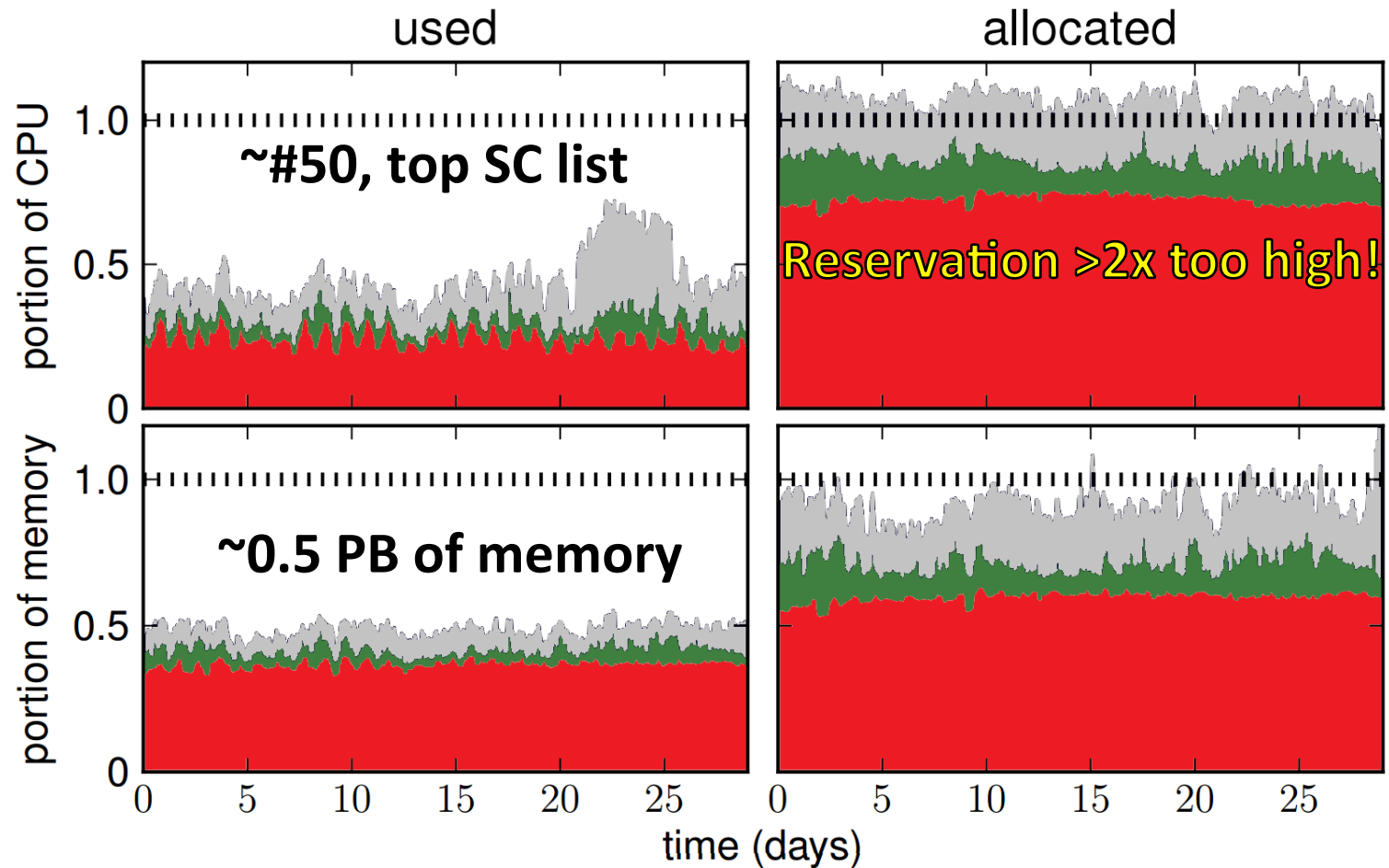
Time-sharing a core

- Memcached runs often enough to keep cache warm
- TLB flush has minimal impact

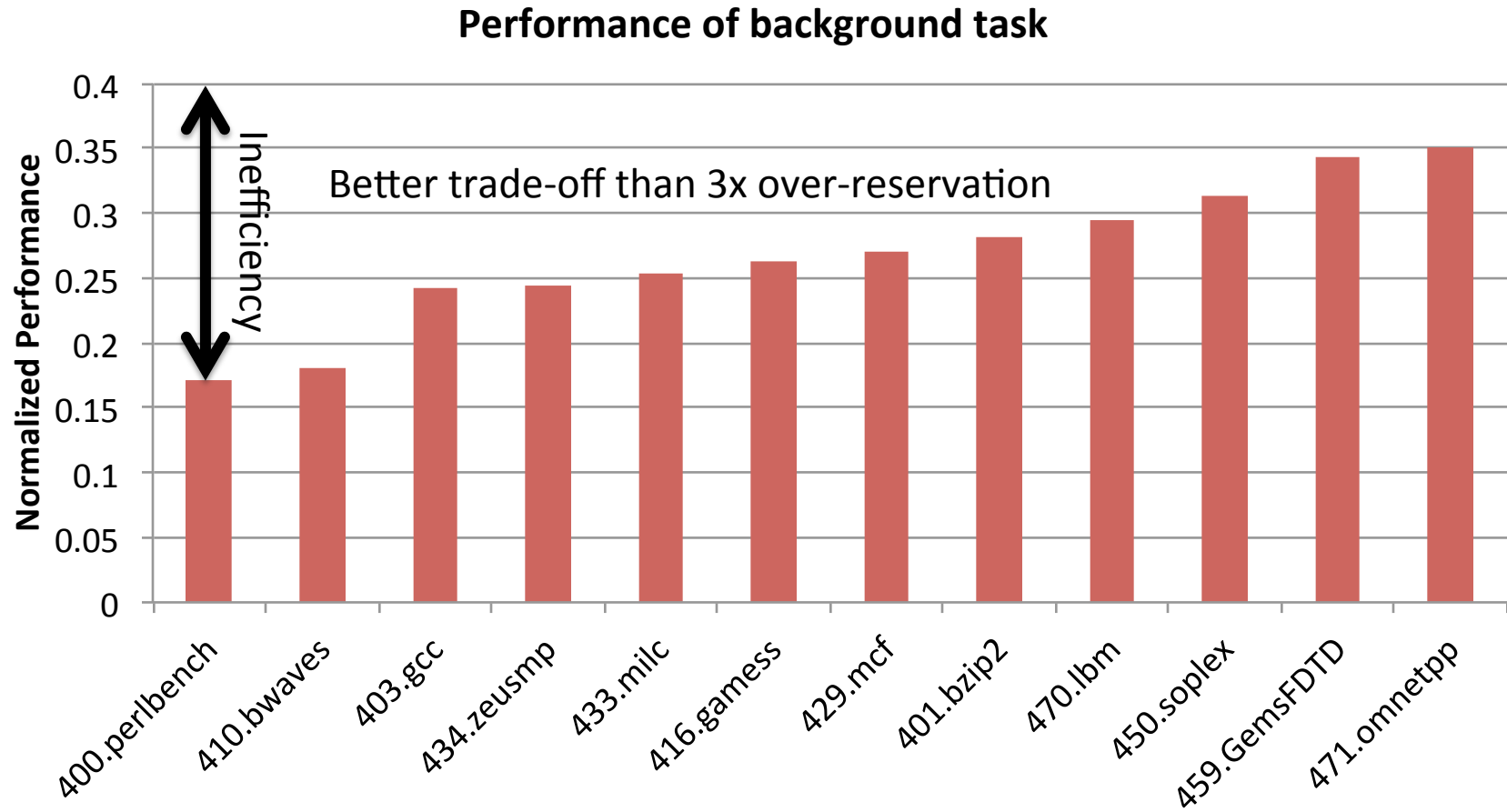


12,000-node Google cluster

[Reiss, SOCC'12]



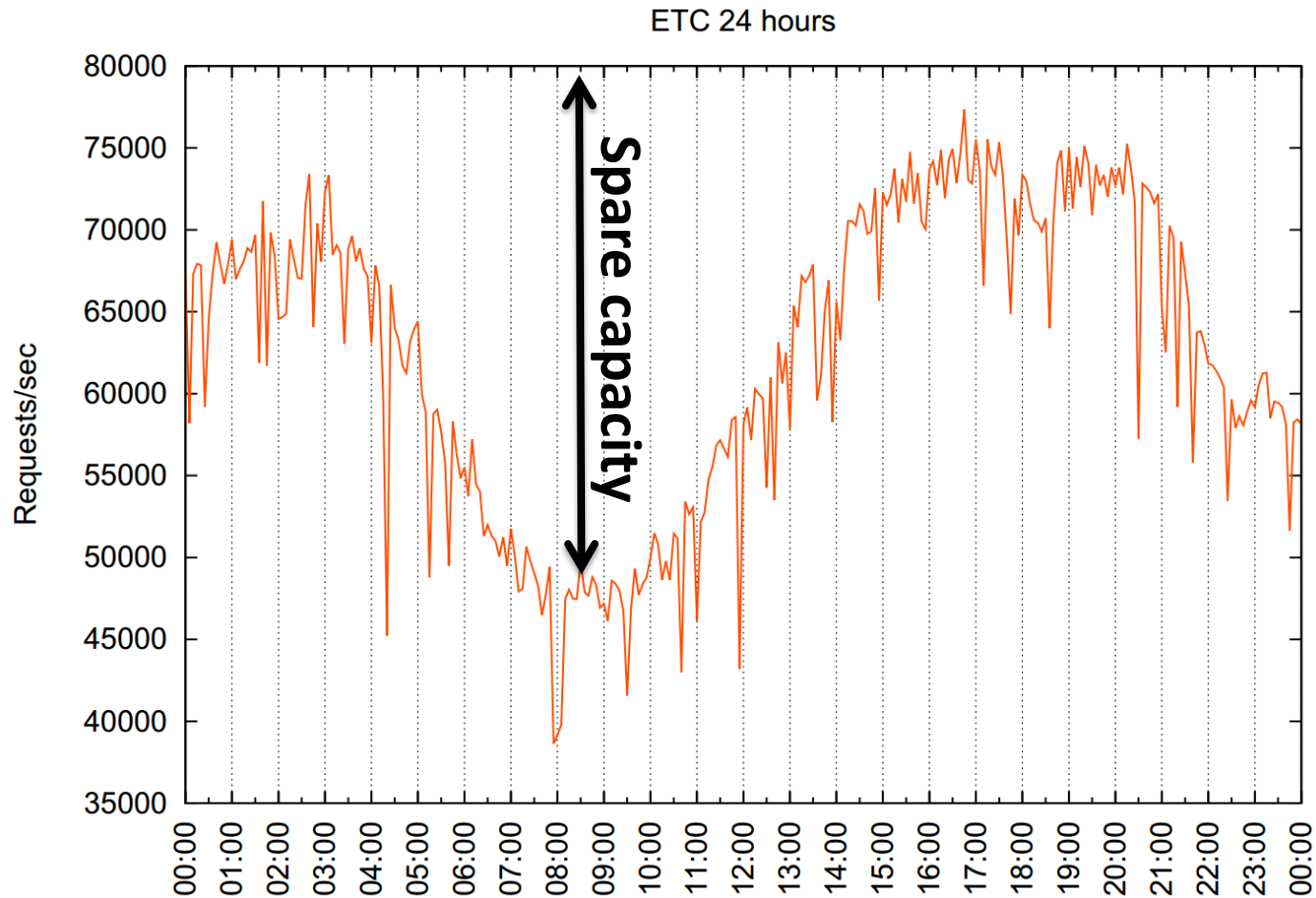
Co-scheduled task incurs the overhead



Low utilization outside Google

- Mozilla: 10%
- McKinsey '08: Industry-wide ~6%
- Gartner '10: Industry-wide 12%
- Tata [HPCA'10]: 12.5%
- Amazon [Liu CGC'11]: 7.3%

Diurnal variation at Facebook [Atikoglu'12]



Queuing Delay

- Systemtap to trace kernel/memcached at runtime
 - We track individual connections to record per-request latency

Who	What	Low load	Overload
Server	RX	0.9us	
	TCP/IP	4.7us	
	epoll() return	3.9us	
	<i>libevent</i>	2.4us	
	read() call/ret	2.5us	
	<i>memcached</i>	2.5us	
	write()+TX	4.6us	
	Total	21.5us	
Client	End-to-End	49.8us	6011us

Queuing Delay

- Systemtap to trace kernel/memcached at runtime
 - We track individual connections to record per-request latency

Who	What	Low load	Overload
Server	RX	0.9us	1us
	TCP/IP	4.7us	4us
	epoll() return	3.9us	2778us
	libevent	2.4us	3074us
	read() call/ret	2.5us	5us
	memcached	2.5us	2us
	write()+TX	4.6us	4us
	Total	21.5us	5872us
Client	End-to-End	49.8us	6011us

Queue #1: Epoll ready list

Queue #2: Libevent
internal event list

Latency with heavy L3 interference

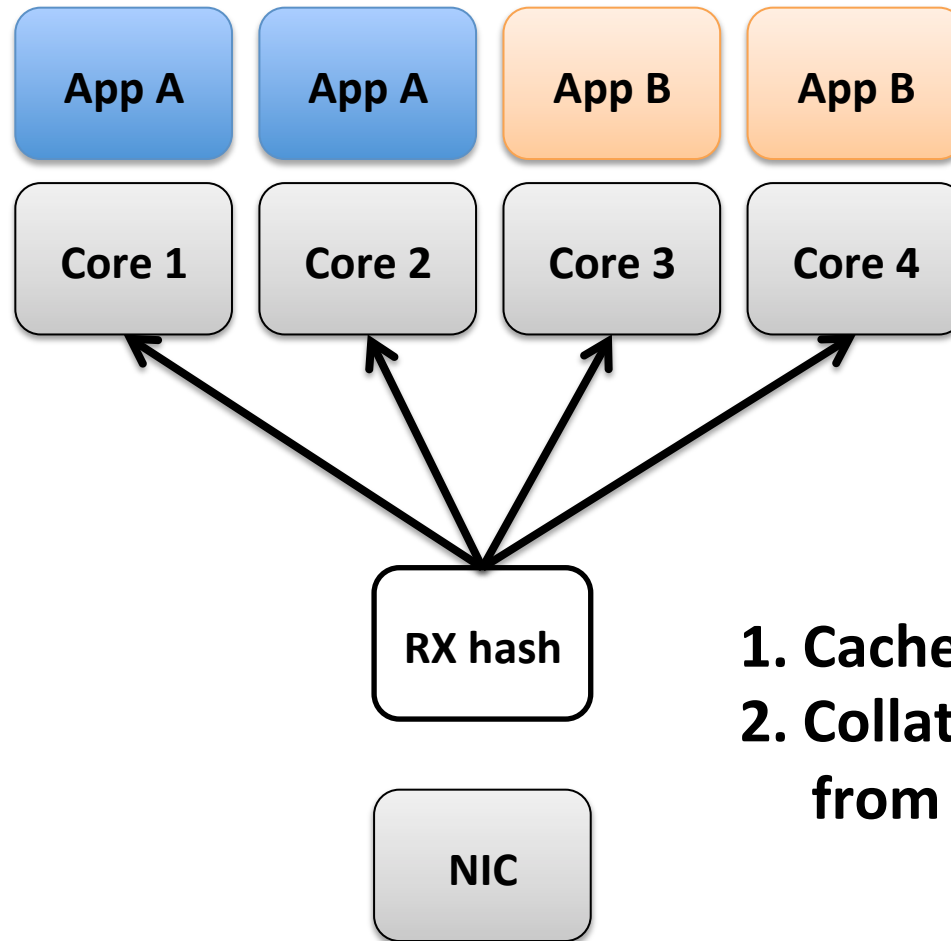
Who	What	Low load	Overload	+ L3 Int.
Server	RX	0.9us	1us	1us
	TCP/IP	4.7us	4us	4us
	epoll() return	3.9us	2778us	3780us
	libevent	2.4us	3074us	4545us
	read() call/ret	2.5us	5us	7us
	memcached	2.5us	2us	4us
	write()+TX	4.6us	4us	5us
	Total	21.5us	5872us	8349us
Client	End-to-End	49.8us	6011us	8460us
	TX-to-RX	36us		
Switch	RX-to-TX	3us		

Systemtap

- Kprobe-based dynamic instrumentation of Linux kernel (like DTrace)

```
probe kernel.function(  
    "tcp_rcv_established@net/ipv4/tcp_input.c"  
) .return {  
    timestamps[$sk, "tcpip_rx"] = gettimeofday_ns()  
}
```

Multi-queue NICs: Receive-Side Scaling



1. Cache misses and IPI
2. Collateral damage from interrupt

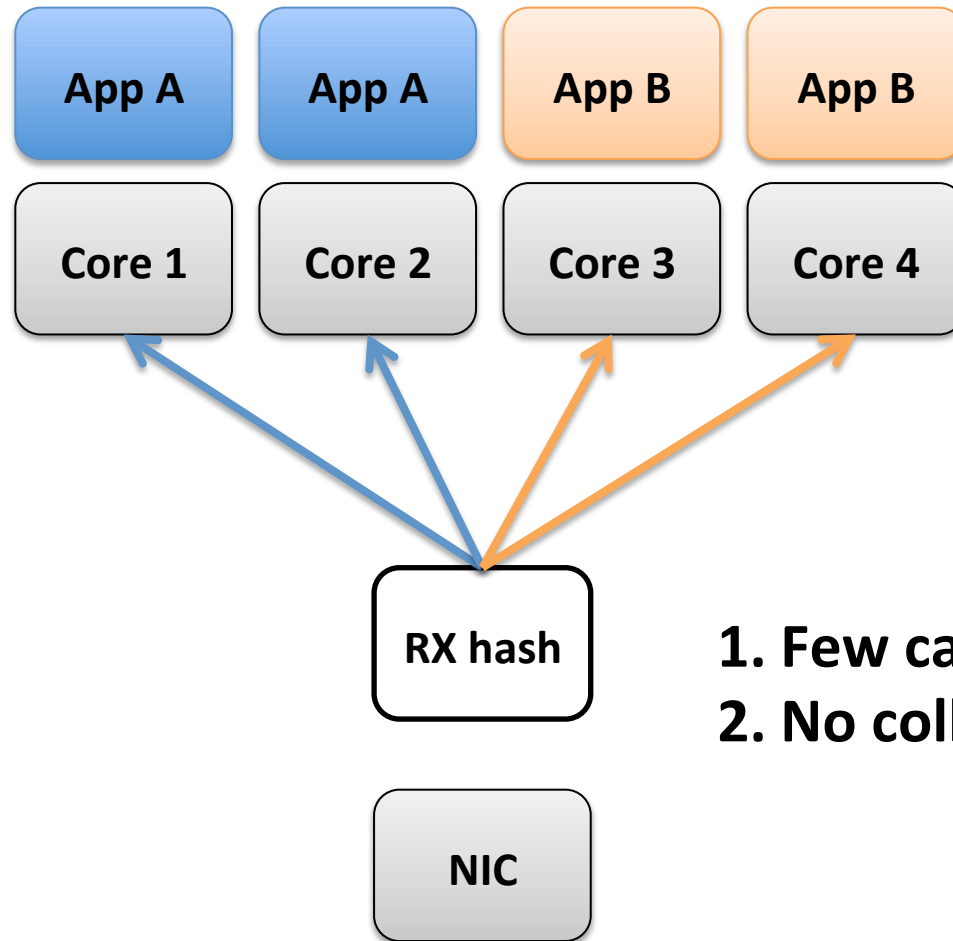


Network interrupt interference

NIC Configuration		Network Traffic	Memcached QPS loss due to concurrent network traffic
		1 Gbps	
Receive-side scaling (rx hashing)		24%	

- Memcached on socket #1, D-ITG on socket #2
- Multi-queue NICs with receive-side scaling spray interrupts across all cores
 - Necessary to handle 10GbE packet rates
 - Can cause massive interference, even at low B/W

Receive-Flow Steering



1. Few cache misses
2. No collateral damage



Receive-flow steering

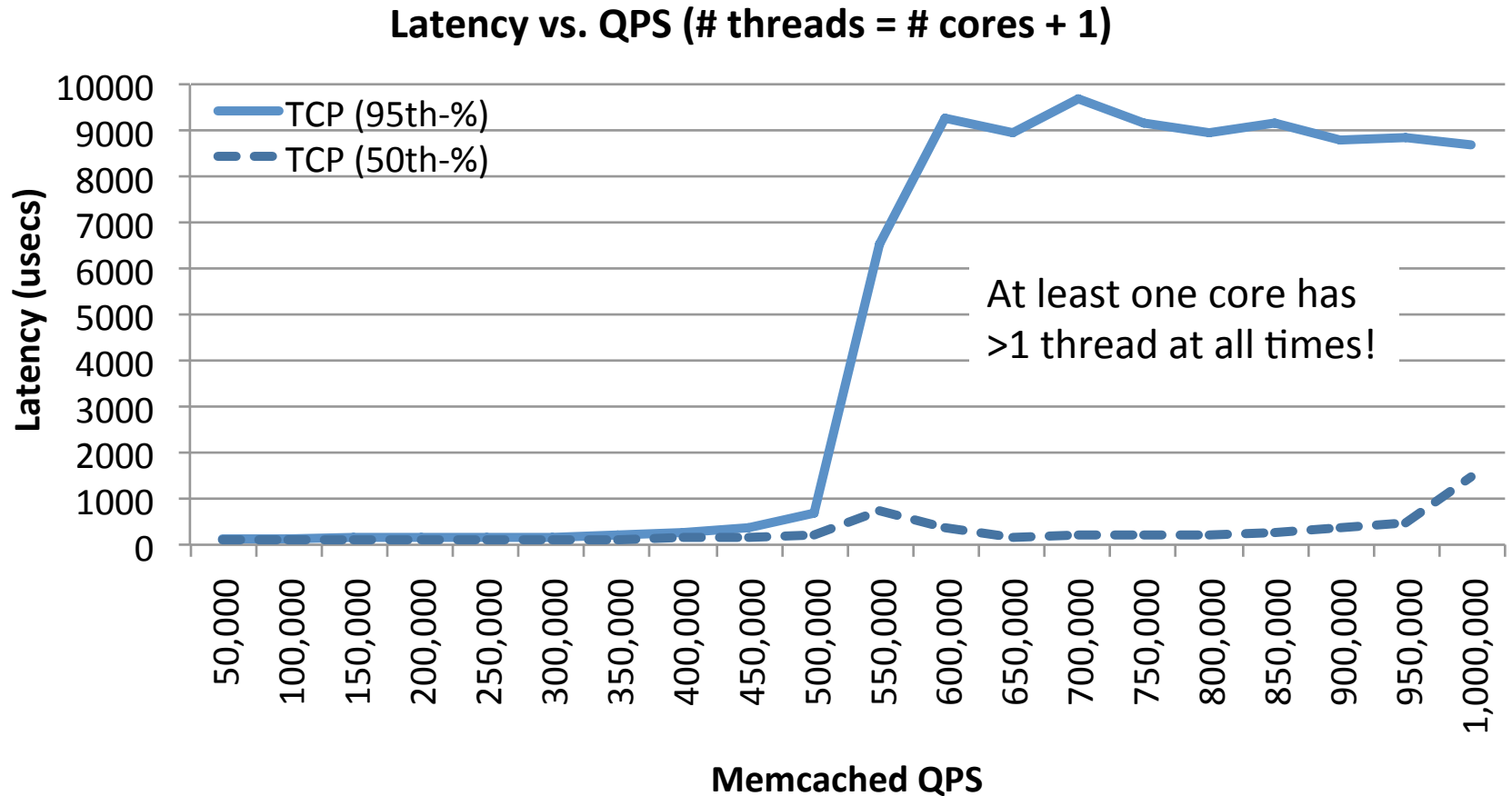
NIC Configuration	Network Traffic	Memcached QPS loss due to concurrent network traffic
	1 Gbps	
Receive-side scaling (rx hashing)	24%	
Receive-flow steering (rx follows tx)	1%	

- Largely mitigated by receive-flow steering
 - Send interrupts to the core responsible for a flow
 - QoS benefits unreported in literature
- Available in Mellanox and Intel NICs
 - Intel's IXGBE driver implements "RX follows TX" policy (not in mainline Linux)

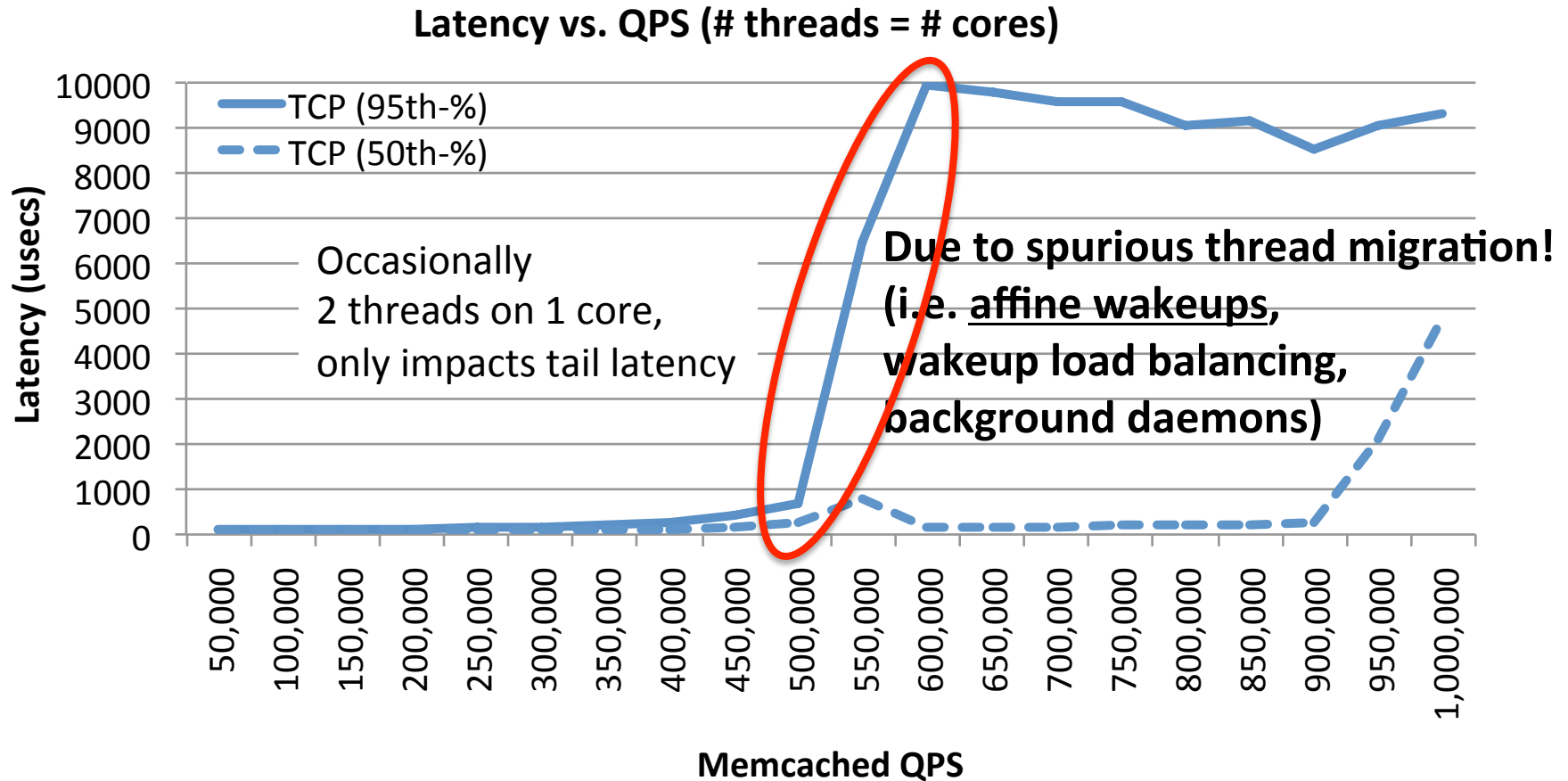
Problems with static connection assignment

- Queuing delay due to...
 - Unbalanced # of clients
 - Hot/cold clients
 - Unbalanced CPU performance (i.e. cache int.)
- Scheduling delay due to...
 - Co-scheduled work
 - Mismatched # of threads and CPUs
- Tail latency suffers if only one thread affected!

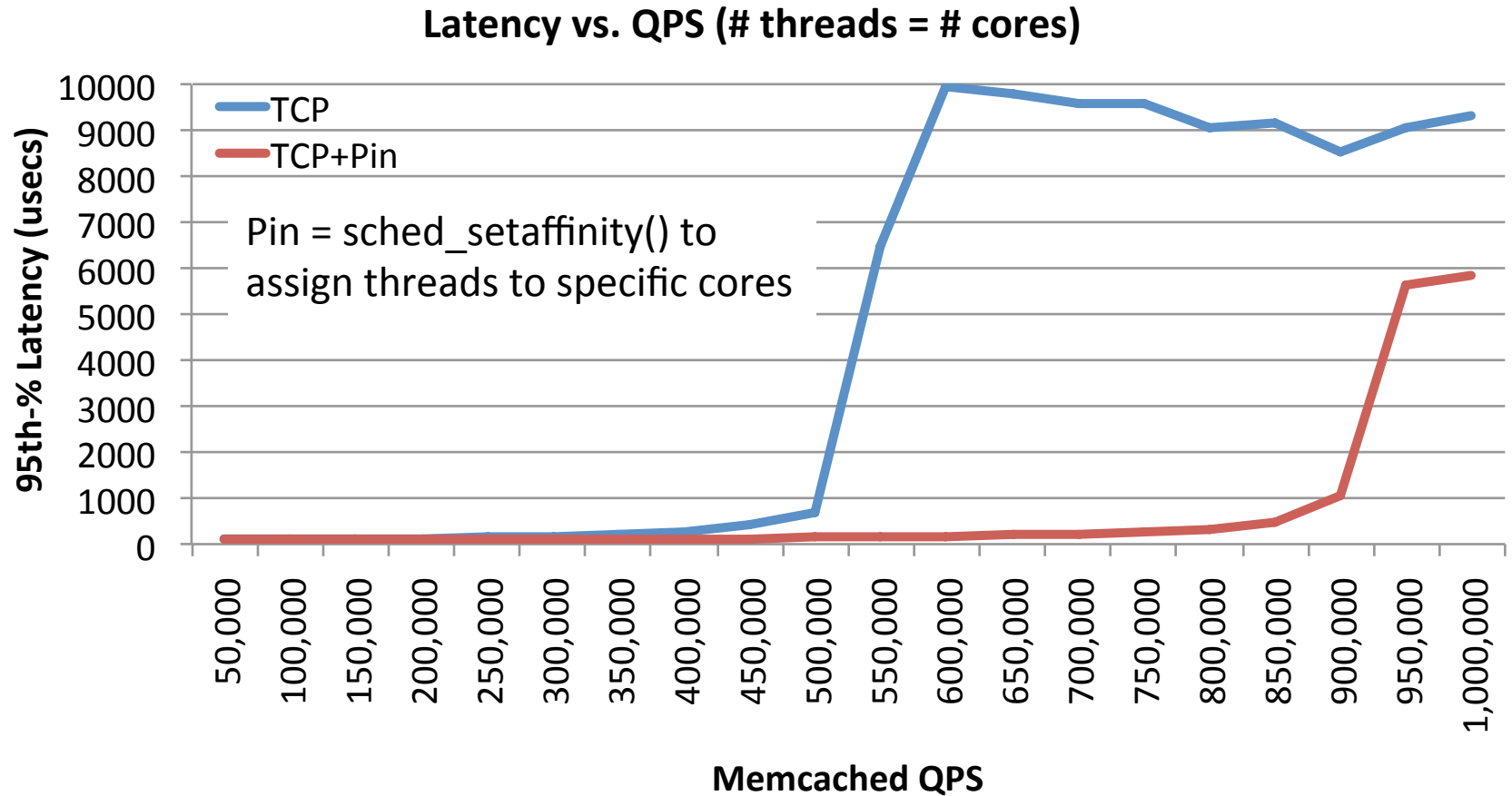
Latency with # threads = # cores + 1



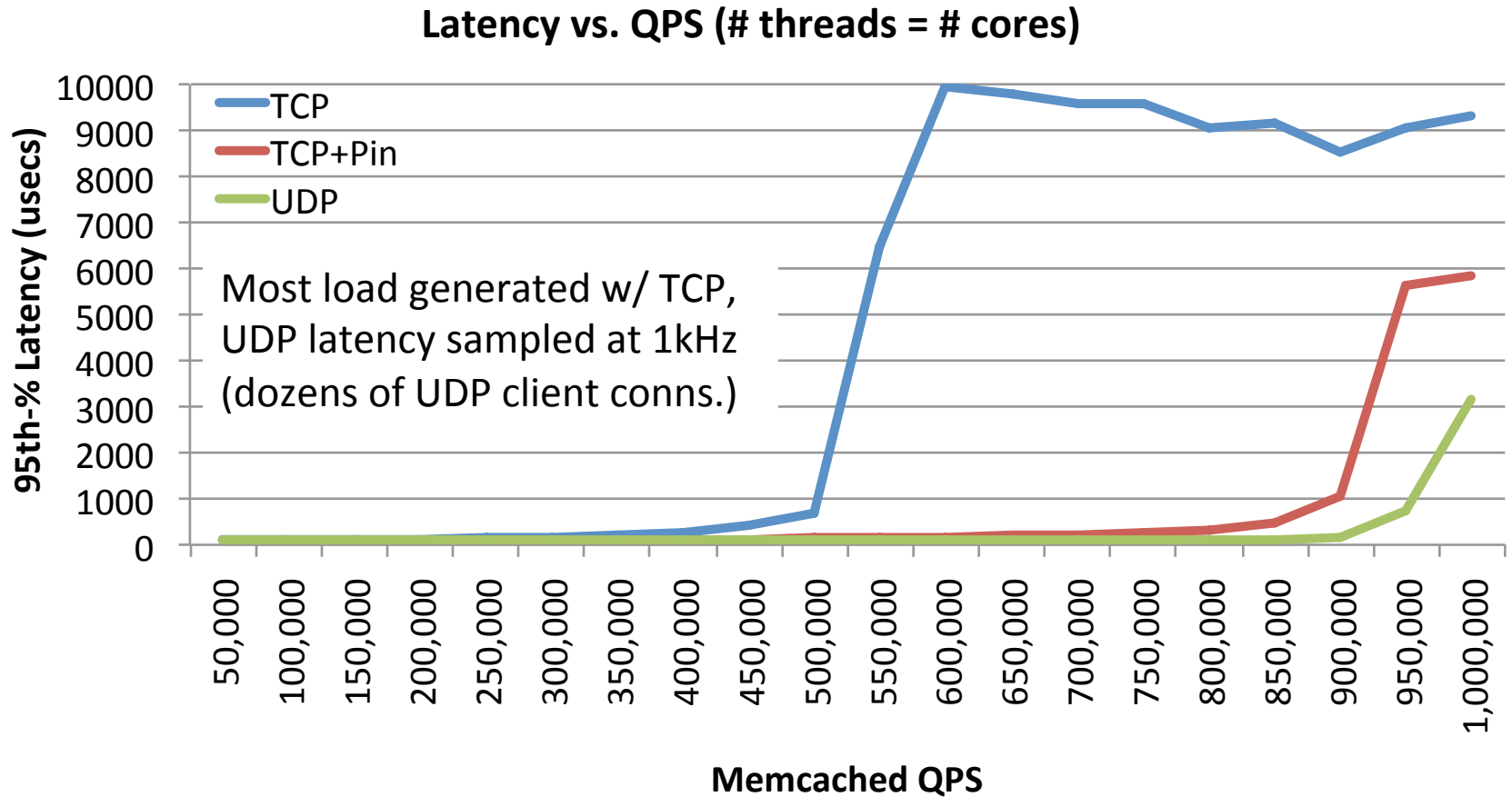
Same result with # threads = # cores!



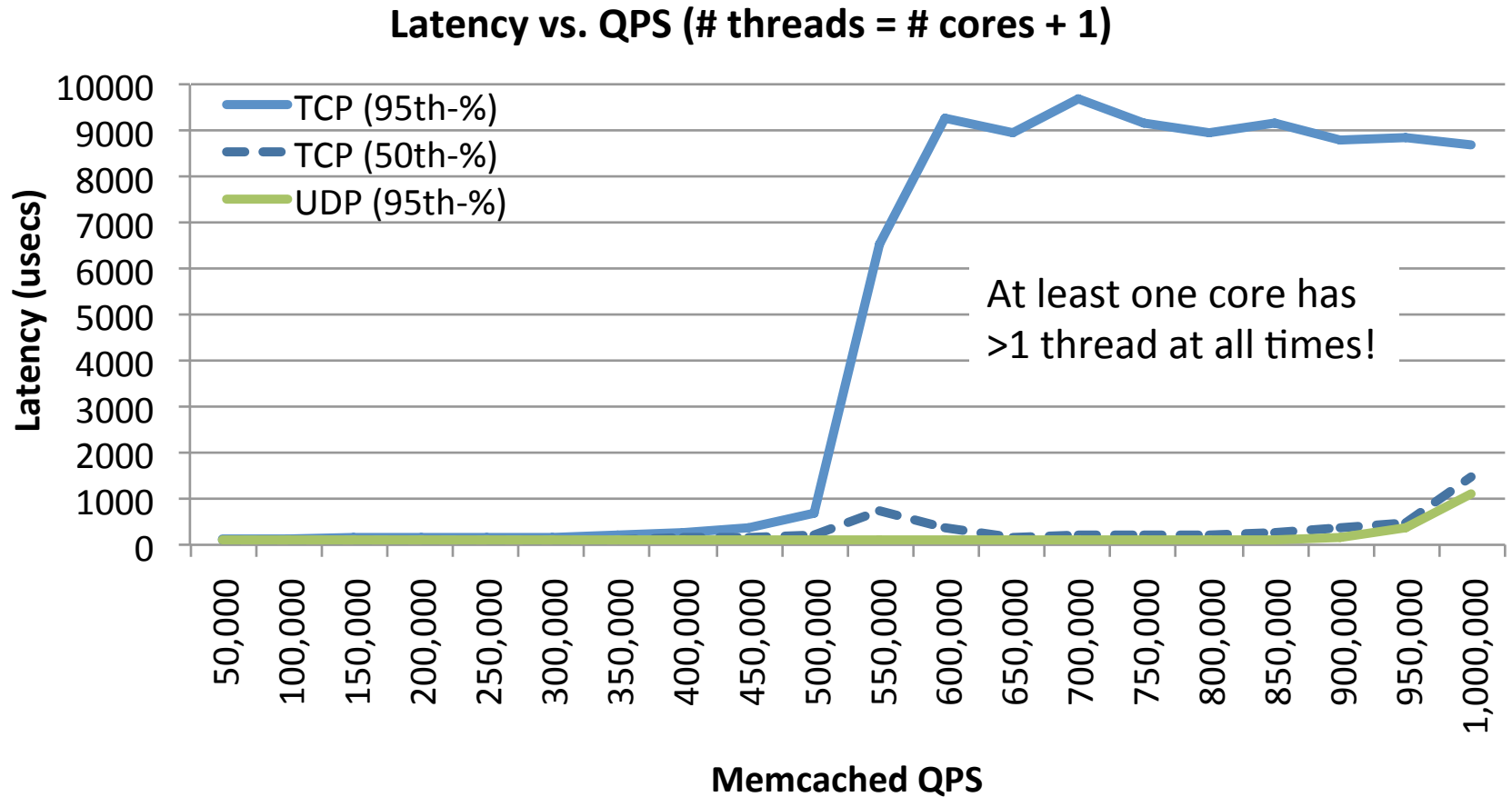
Pin threads to prevent migrations



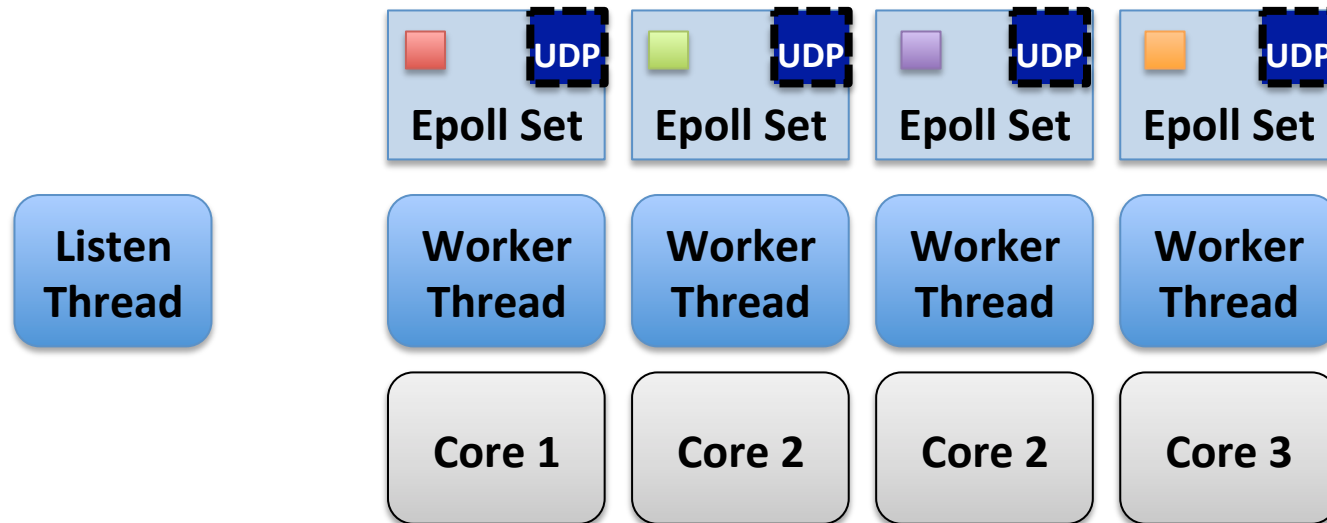
Interesting UDP behavior



Interesting UDP behavior



Memcached UDP handling



- Dynamically load balanced!
- Poor QPS due to lock contention on UDP socket
 - Multiple UDP sockets might reintroduce QoS problem

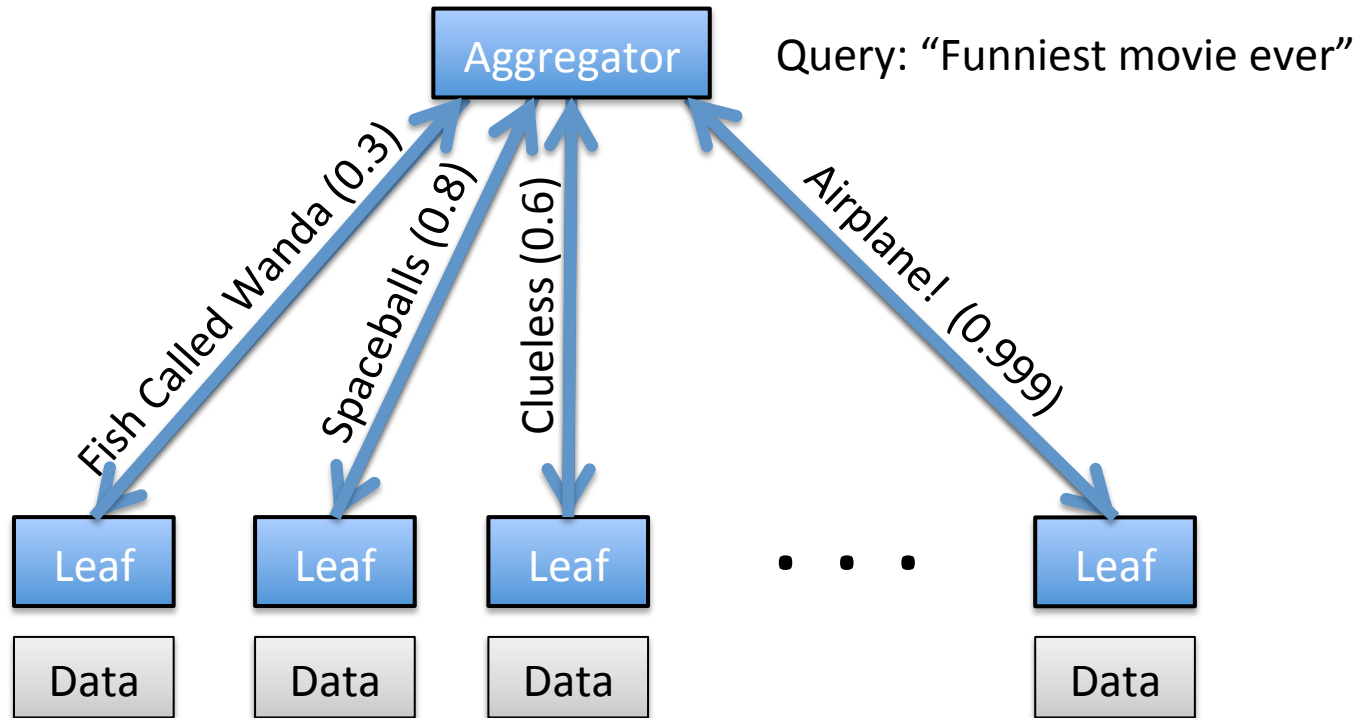
More Related Work

- Deadline scheduling
 - Liu and Layland [JACM'1973], seminal results
 - AQuoSA [SPE'08], EDF w/ dynamic reservations, SCHED_DEADLINE
 - **We should schedule events, not processes!**
- Ultra-low latency services
 - RAMCloud [OSR'10], Chronos [SOCC'12]
 - User-level networking, bypass OS!
 - **Must reconcile provisioned vs. nominal QPS**

Consolidation related work

- Don't run interfering jobs together
 - Bubble-Up [Mars, MICRO'11]
 - Paragon [Delimitrou, ASPLOS'13]
 - **Don't address the root cause of QoS problems**
- Network QoS: HULL [NSDI'12]
- Hardware interference mitigation
 - Vantage [Sanchez, ISCA'11]
 - SMT QoS [Herdreich, HotPar'12]
 - **Reduce amount of interference, don't eliminate**

Partition/Aggregate Workloads

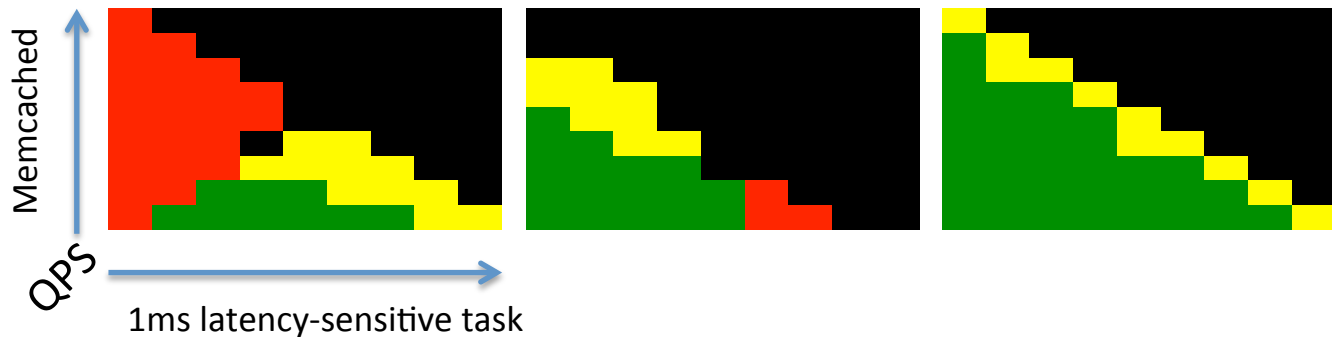


Overall latency dominated by the "tail"
Long latency = lost revenue

Other schedulers do better

		Completely Fair Scheduler	POSIX Realtime + Bandwidth Limit	BVT [Duda'99] + Grace Period
Feature	Reservations			
	Configurable preemption			
	Work conserving			

Achieved QoS with 2 low-latency tasks



Alternative event-handling paradigms

- Sockets shared across event sets
 - Be mindful of socket lock contention...
 - Thundering herd; don't put socket in EVERY set
- Event sets shared across threads
 - Edge-triggered or EPOLL_ONESHOT to avoid thundering herd
 - Not supported by libevent
 - Be mindful of **event set** lock contention...
- Connection stealing/load balancing
 - Pisces [Shue et al., OSDI'12]
- Event stealing
 - Maintain thread-safe queue of events returned by `epoll_wait()`
 - Steal events from neighbors when idle
- Comparison = future work 😊

fini