Caladan: Mitigating Interference at Microsecond Timescales




Josh Fried

Zain Ruan

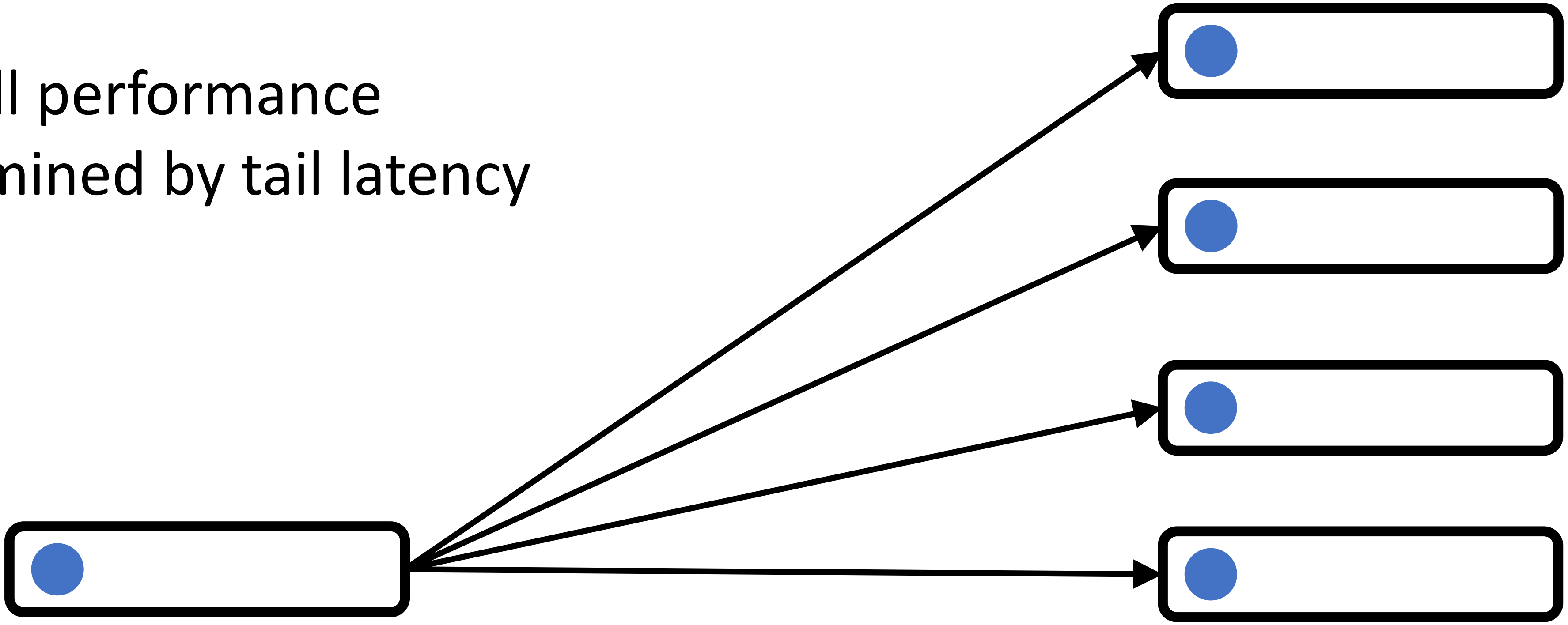
Amy Ousterhout

Adam Belay

CSAIL

Low Tail Latency is Critical in Datacenters

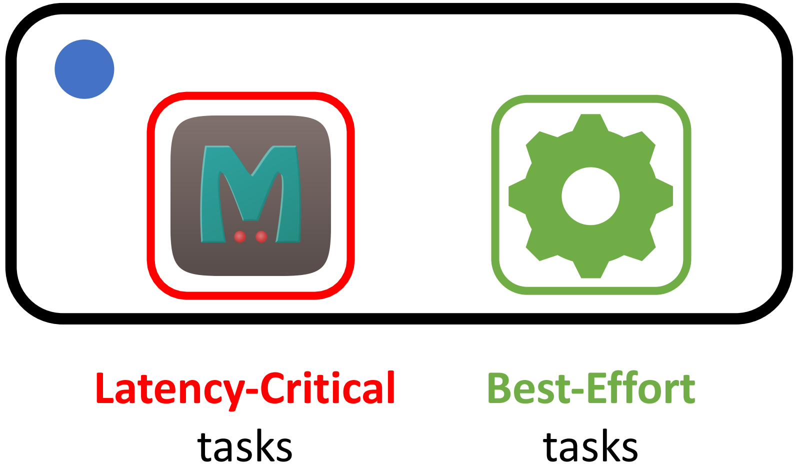
- High degrees of fanout
- Overall performance determined by tail latency



Must Balance Latency with Efficiency

Ideal: Operate hardware at 100% utilization

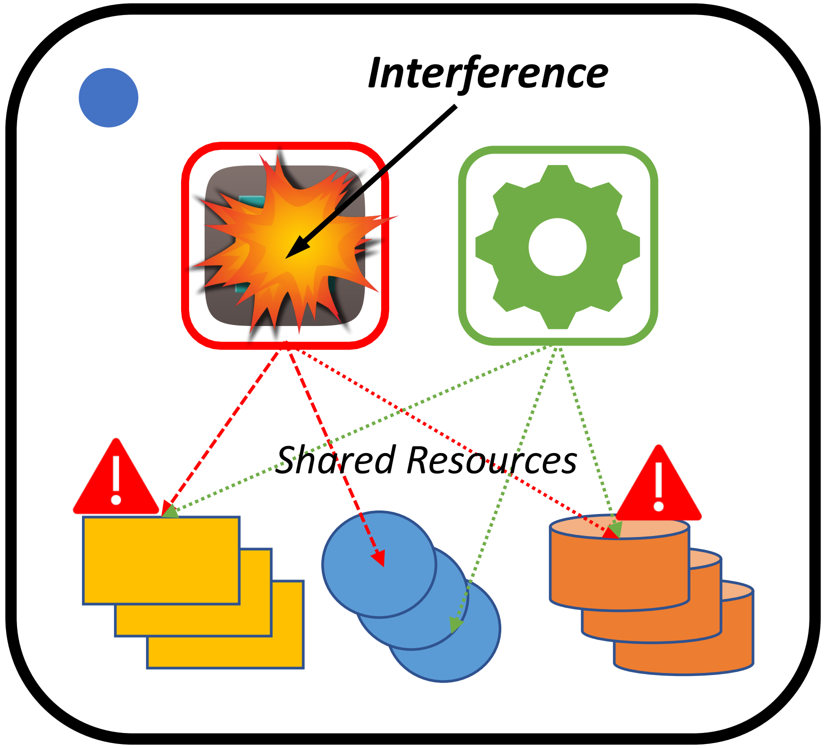
Operators pack multiple tasks on each machine



Challenge: Noisy Neighbors

Tasks on the same CPU contend for shared resources

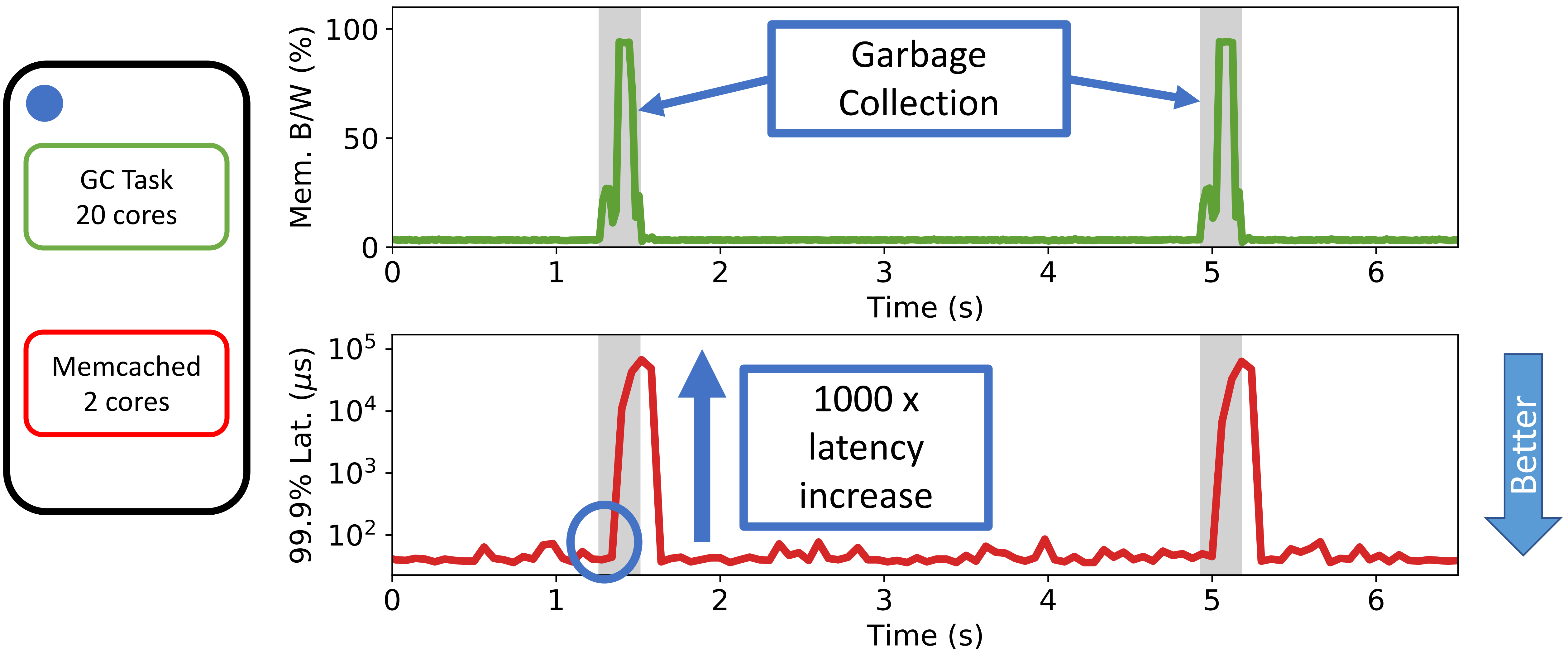
- Cores
- Caches
- Memory bandwidth
- Shared execution units



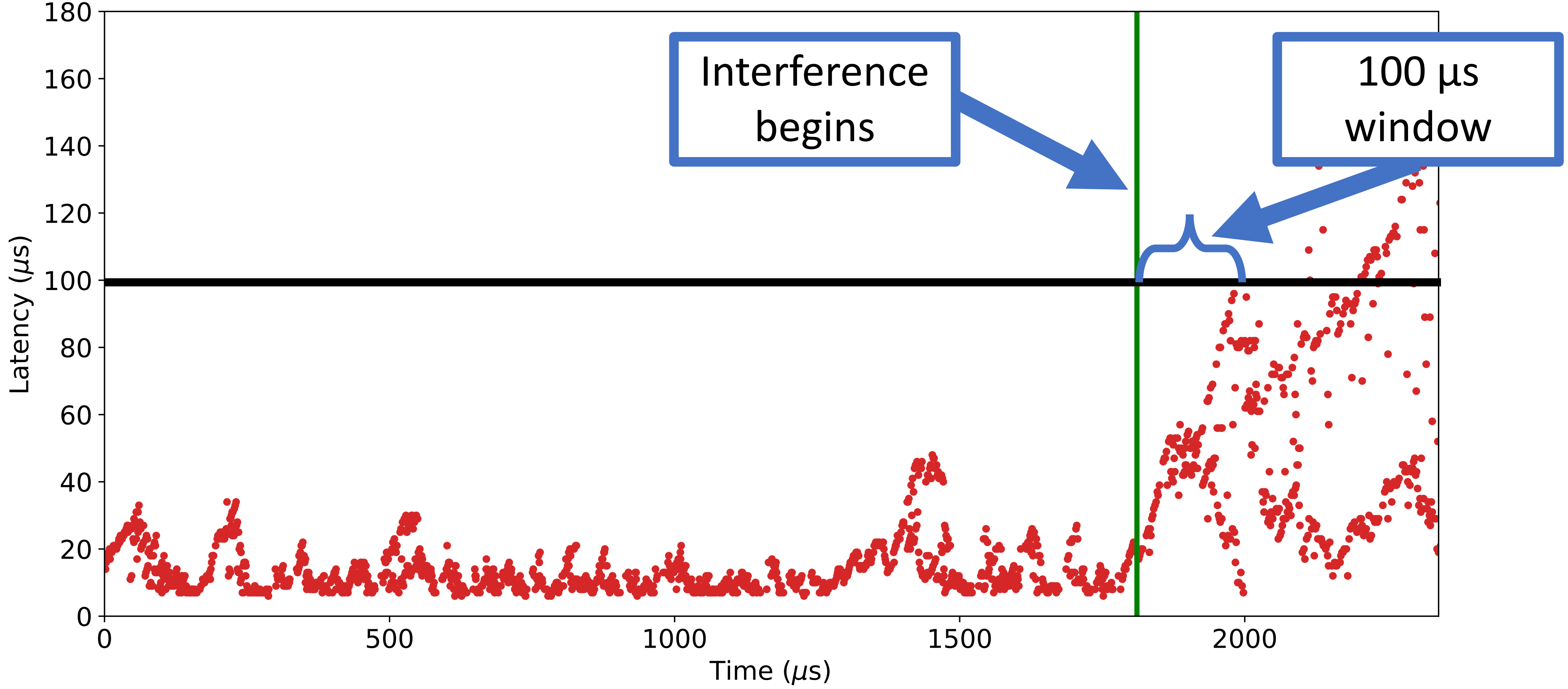
Challenge: Resource Usage Constantly Shifts

- Application load can be bursty at microsecond-scales
  - Network traffic on Google datacenter machines
  - Thread wakeups in Microsoft's Bing service
- Many applications exhibit phased behaviors
  - Compression, compilation
  - Spark compute jobs
  - Garbage collection

Interference Example

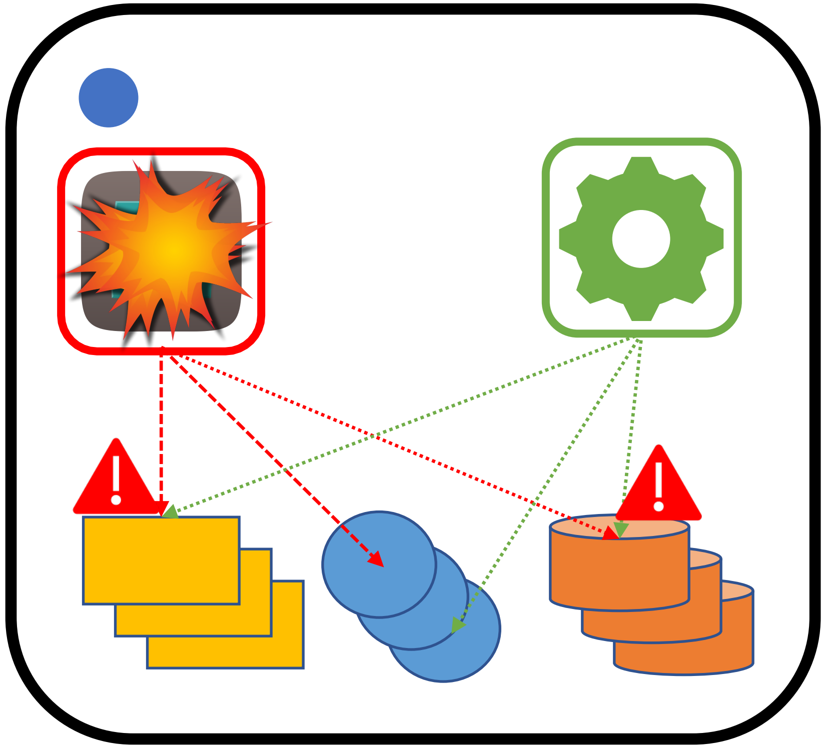


How fast must we react?



100 µs window to react to keep latencies below SLO

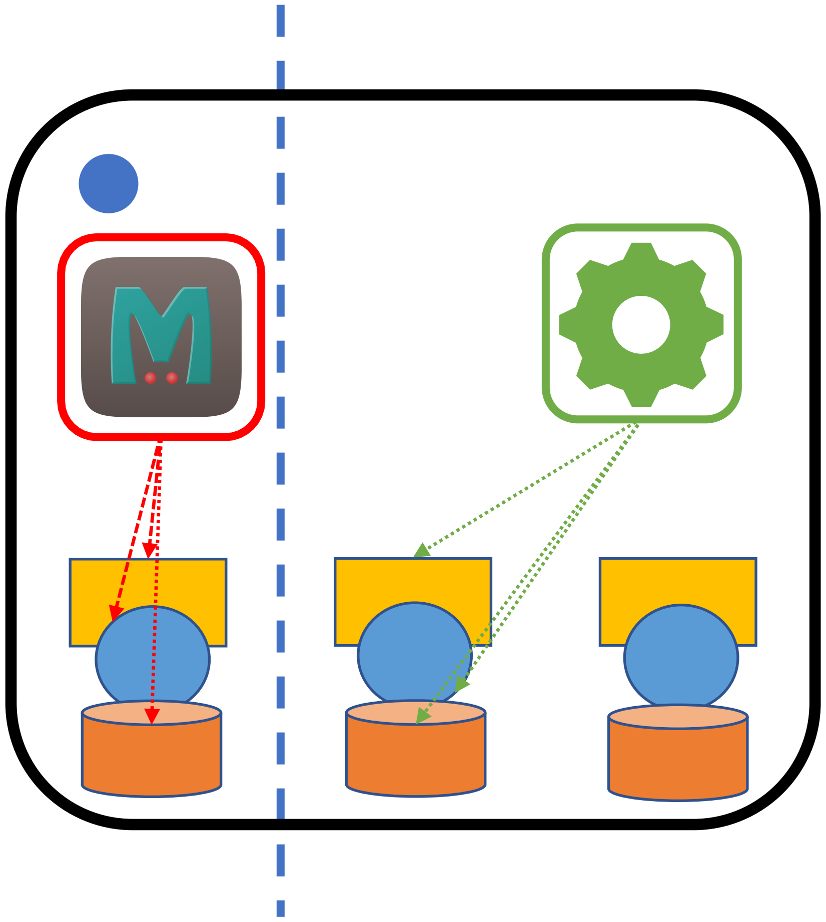
Existing Solutions



Existing Solutions

Existing solutions solve the problem by partitioning resources

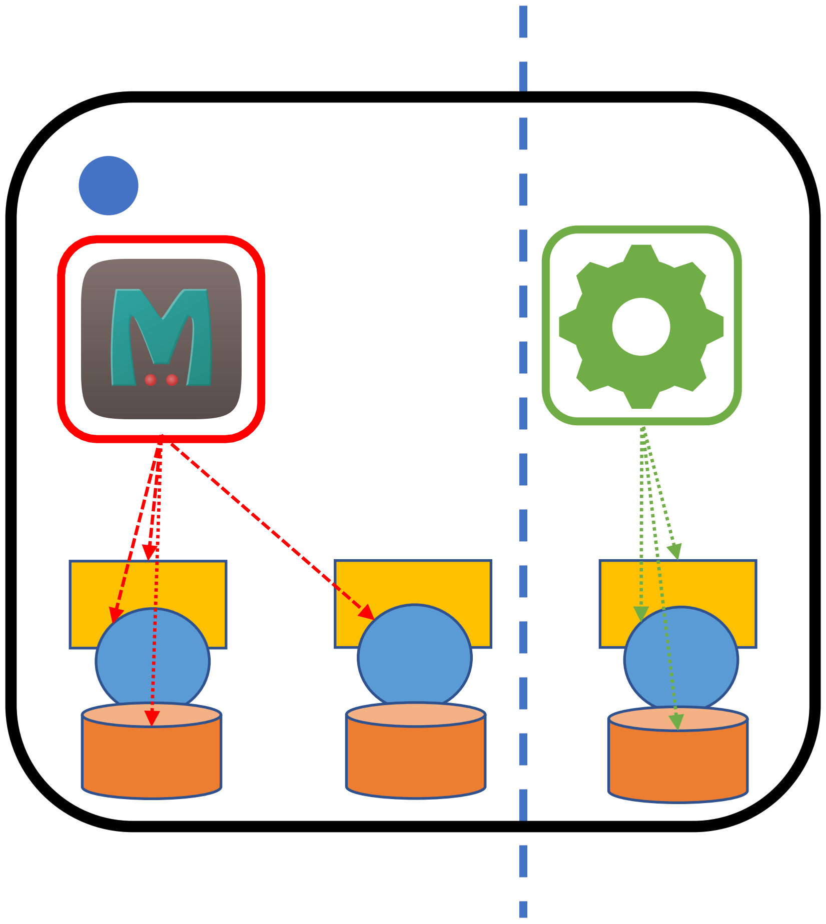
- e.g. dedicating cores, partitioning caches, etc.



Existing Solutions

Existing solutions solve the problem by partitioning resources

e.g. dedicating cores, partitioning caches, etc.



Recent systems dynamically adjust partitions

- Heracles [ISCA ‘15]
  - Converges in 30 seconds
- Parties [ASPLOS ‘19]
  - Converges in 10-20 seconds

100,000x too slow for GC example

Goal

Provide strict **performance isolation** and high **resource utilization** for datacenter servers

Not achievable unless we can detect and mitigate interference at microsecond timescales

Challenges at the μs-Timescale

Finding signals that accurately indicate interference

- Multiple types of interference (LLC, Memory bandwidth, etc)
- Many possible tasks could be causing interference
- Commonly used signals take milliseconds or more to stabilize

Gathering signals and reacting with low overhead

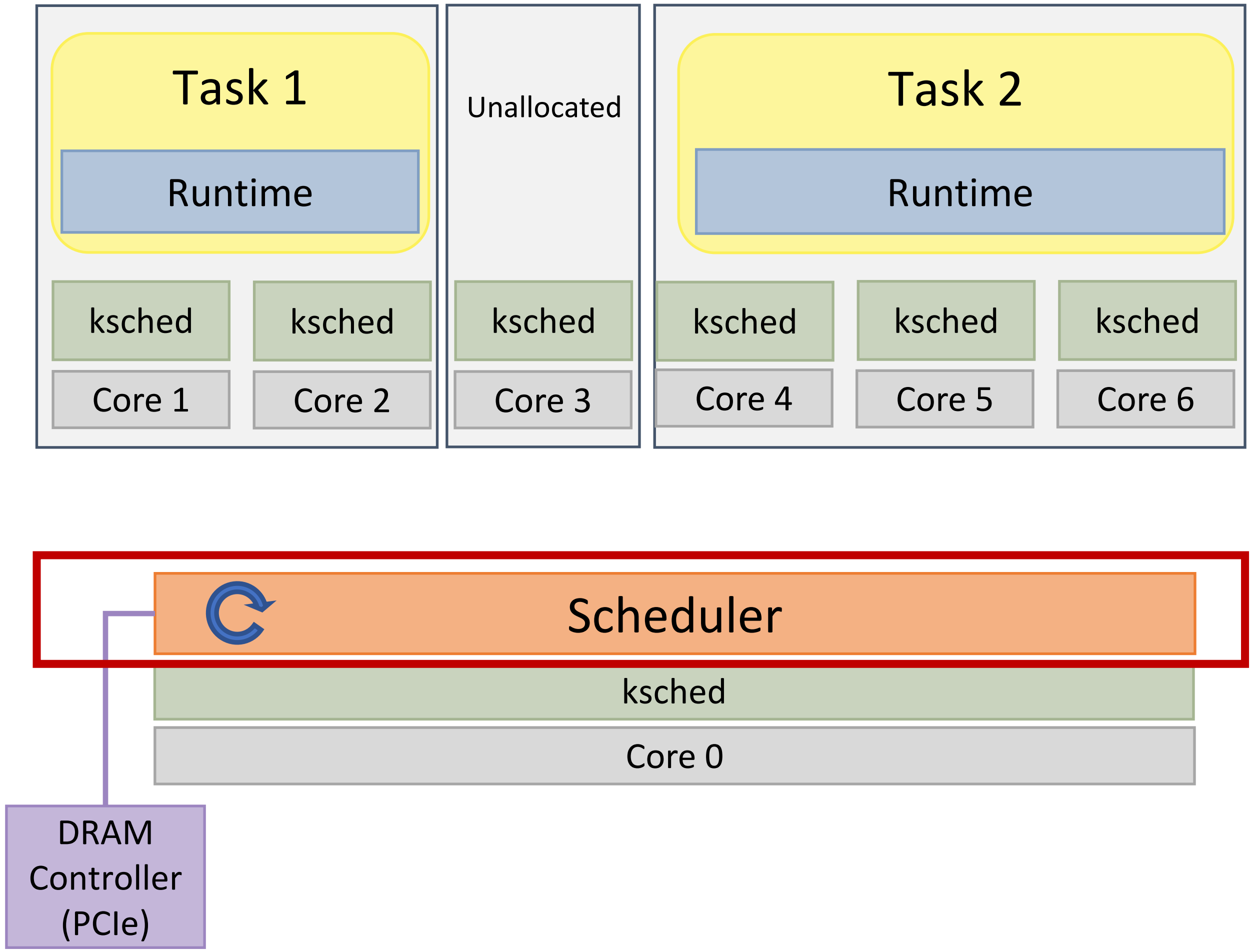
- Existing mechanisms don’t scale well with many cores and tasks

Caladan's Contributions

- Use exclusively core allocation to manage interference
  - Previous systems partition caches, memory bandwidth, etc
- New signals for multiple forms of interference
  - Accurately identify type and source in microseconds
- KSCHED: kernel module to make signal gathering and core allocation scalable
  - Can collect perf counters from all cores in several microseconds

Caladan’s Components

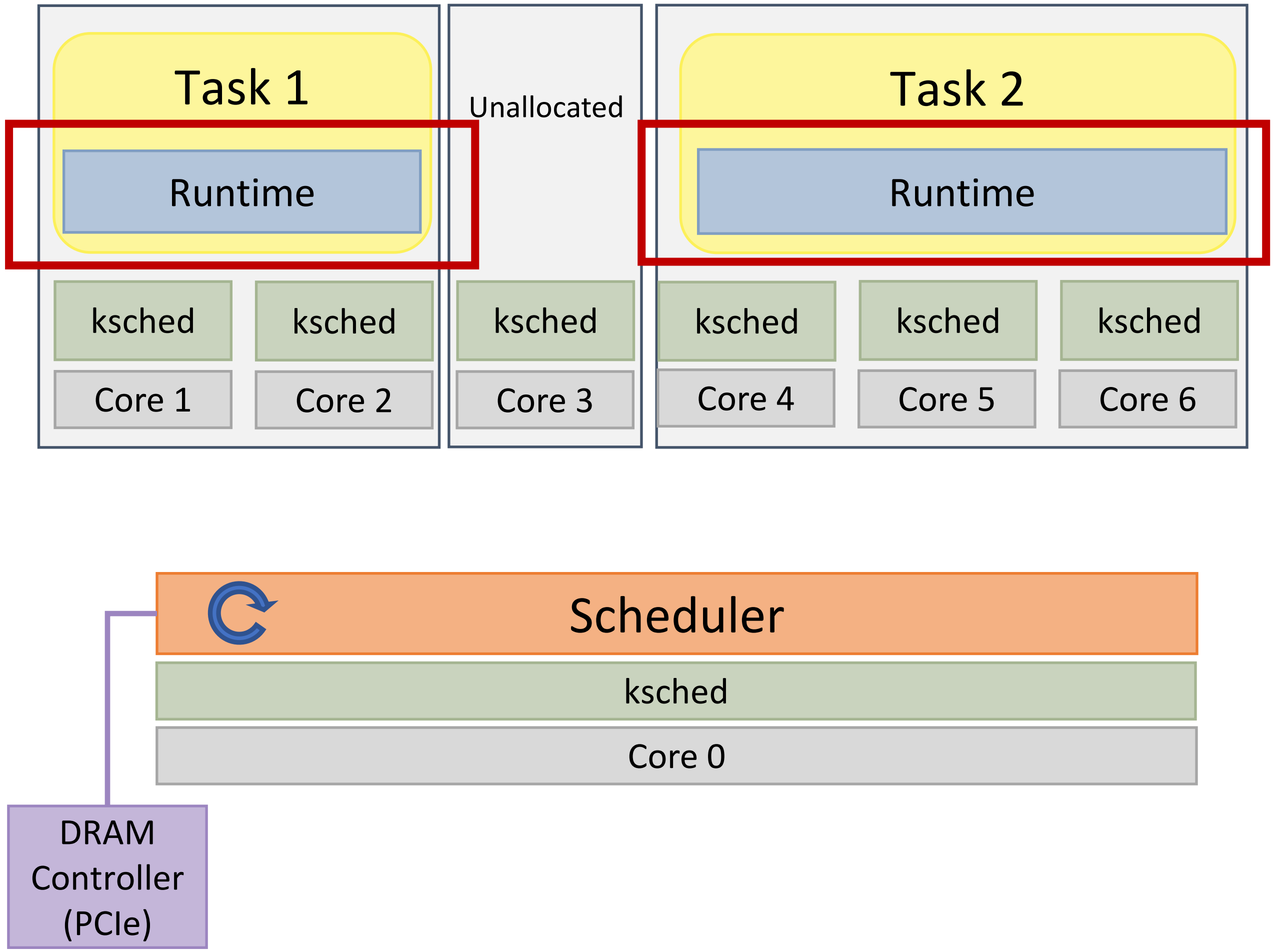
- **Scheduler** core spin polls for signals, assigns tasks to cores



Caladan's Components

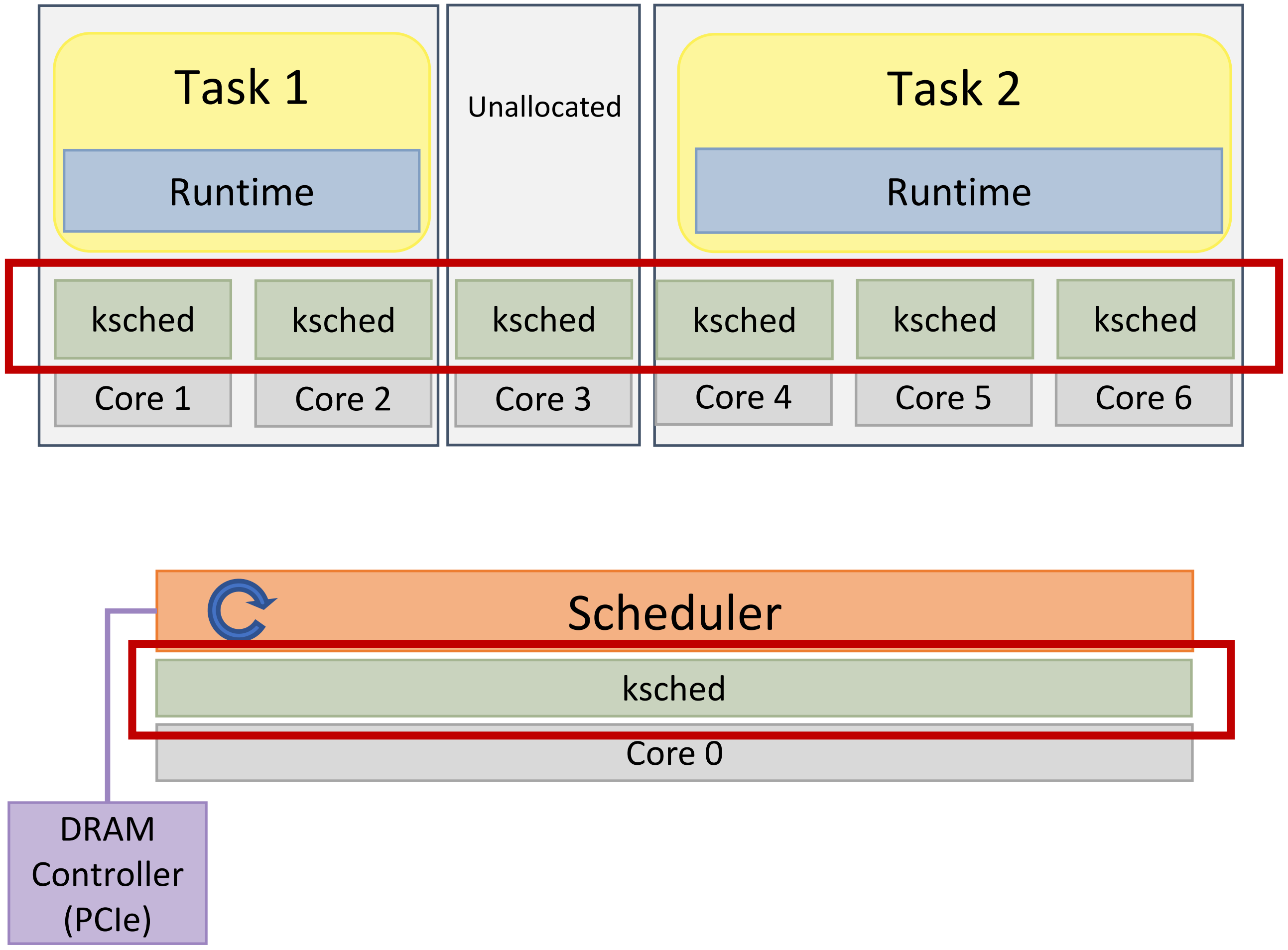
- **Scheduler** core spin polls for signals, assigns tasks to cores

- Tasks link with **runtimes**
  - Provide threading, I/O, etc.
  - Expose signals to scheduler

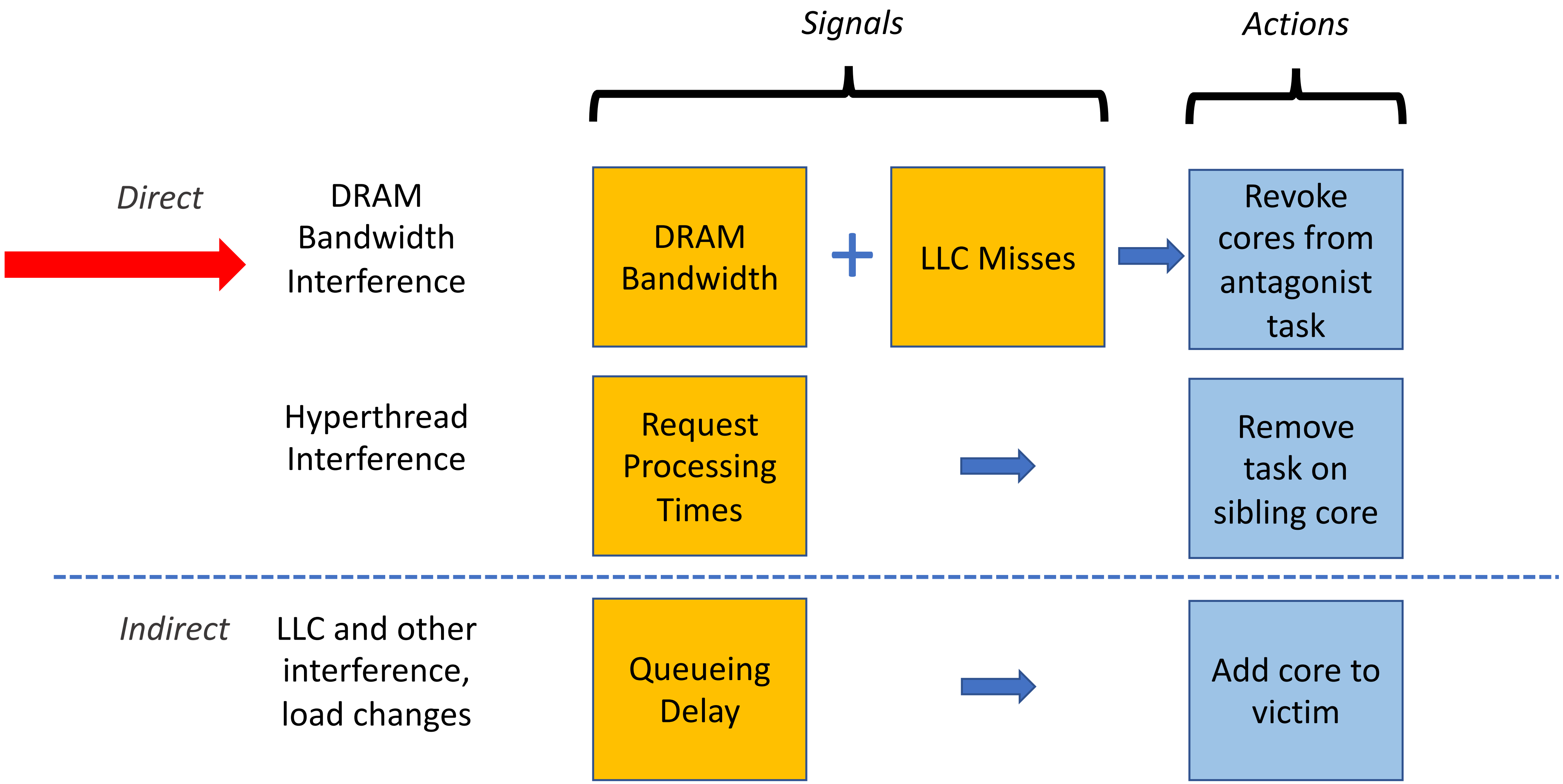


Caladan’s Components

- **Scheduler** core spin polls for signals, assigns tasks to cores
- Tasks link with **runtimes**
  - Provide threading, I/O, etc.
  - Expose signals to scheduler
- **KSCHED** accelerates scheduling and signal gathering



Mitigating Interference



Signal Sources

DRAM controller

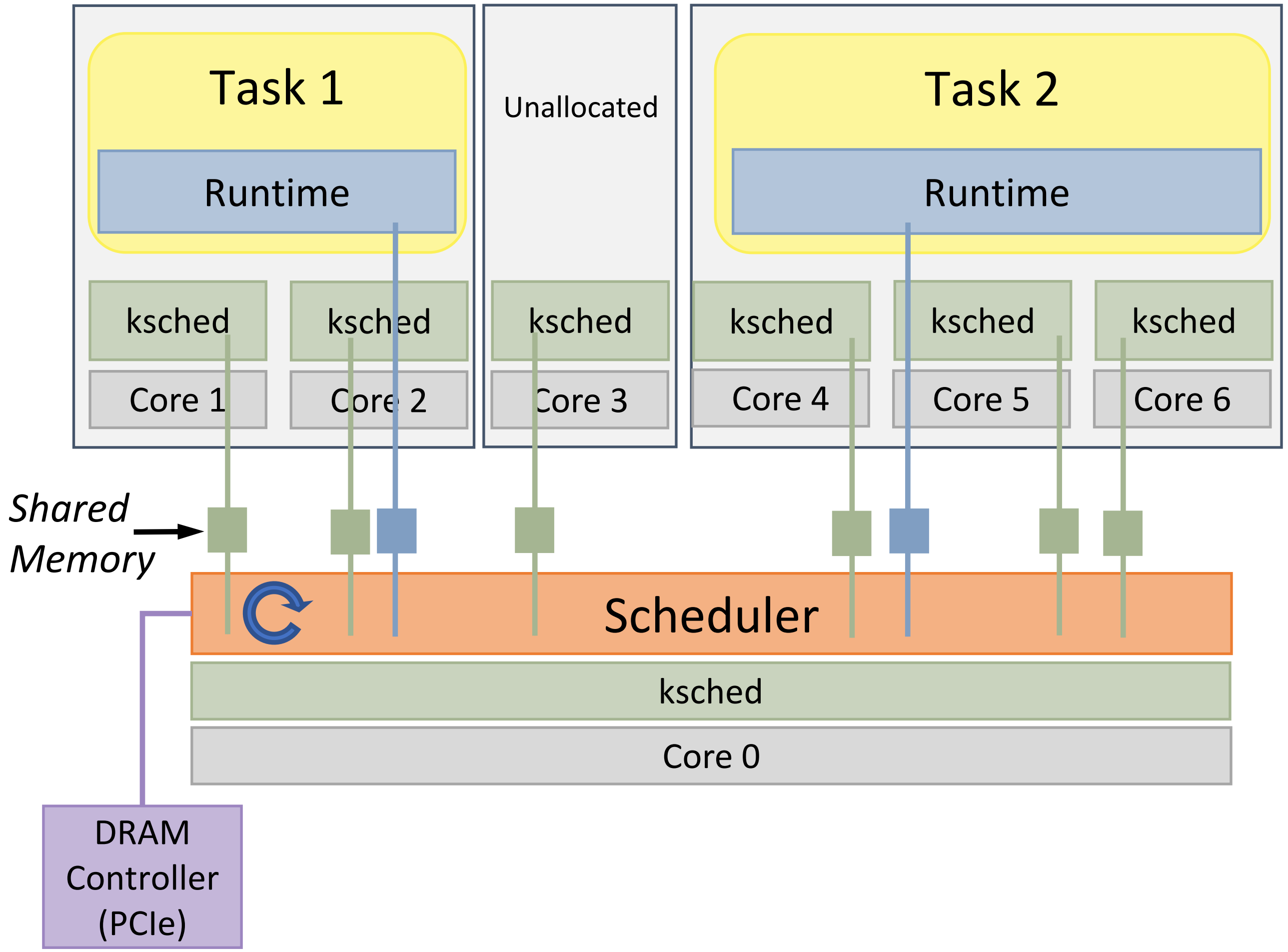
- DRAM bandwidth utilization

Runtime shared memory

- Queueing delays
- Request processing delays

KSCHED shared memory

- per-core LLC miss counters



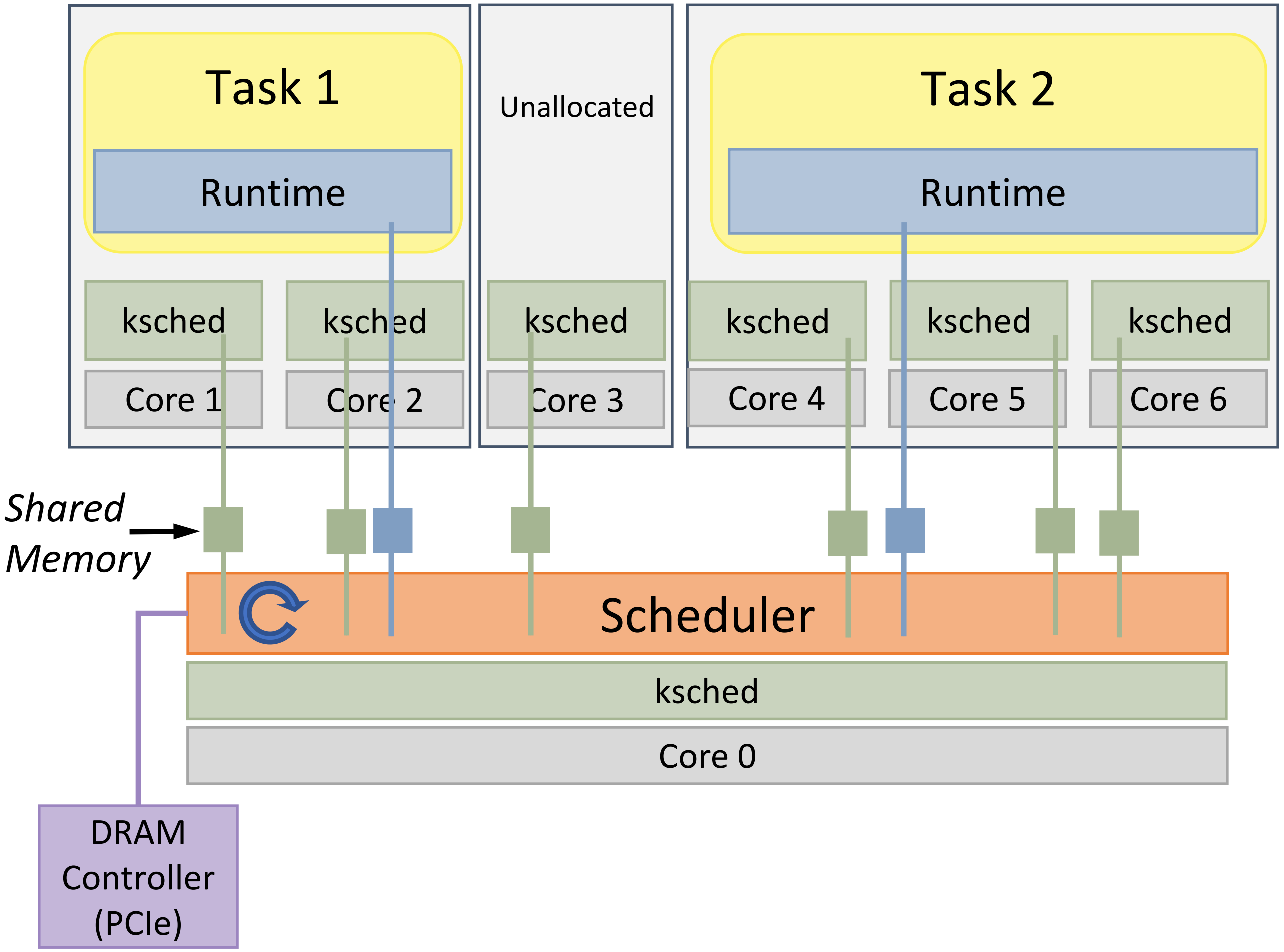
Core Allocation

Existing systems use Linux system calls for scheduling

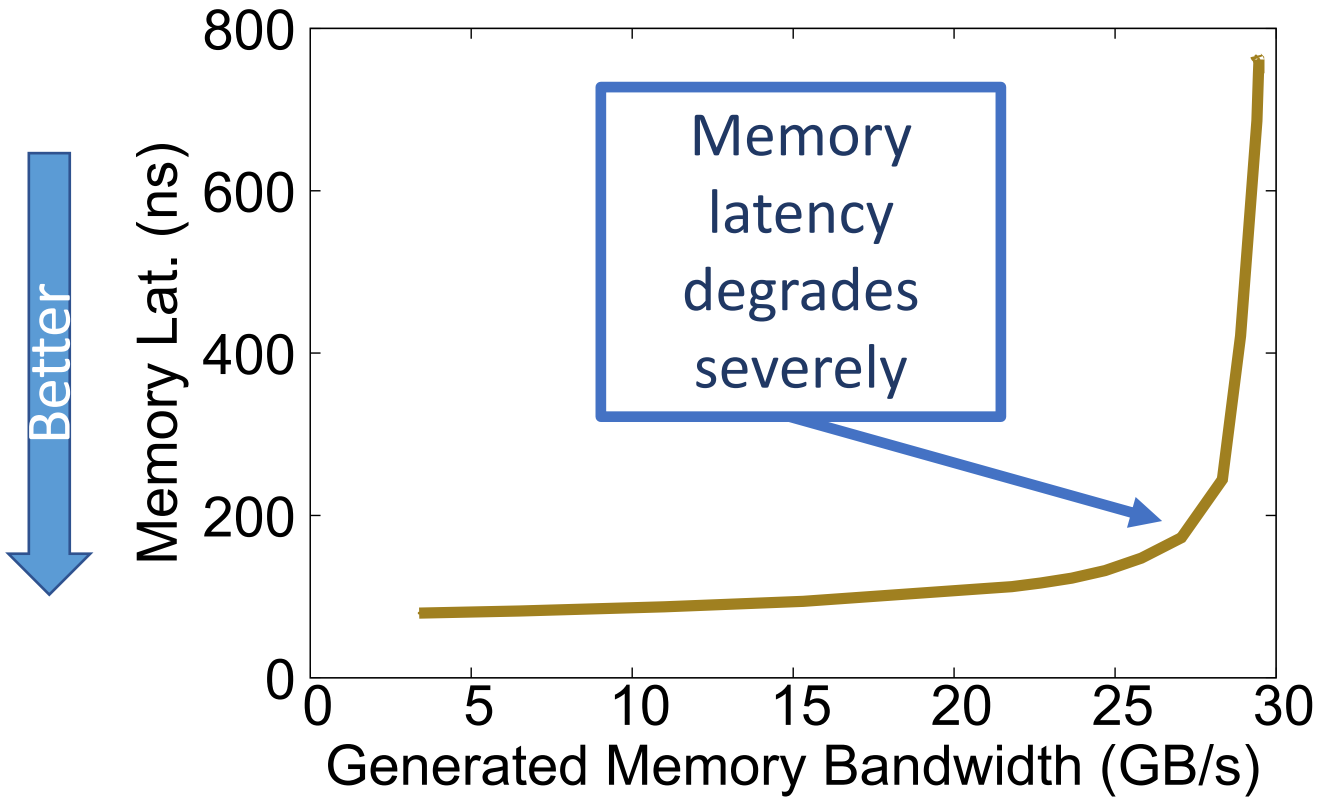
sched_setaffinity(), tgkill(), etc.

KSCHED Optimizations:

- Offload scheduling work
- Multicast Inter-processor Interrupts (IPI)
- Asynchronous interface



Mitigating Memory Bandwidth Interference

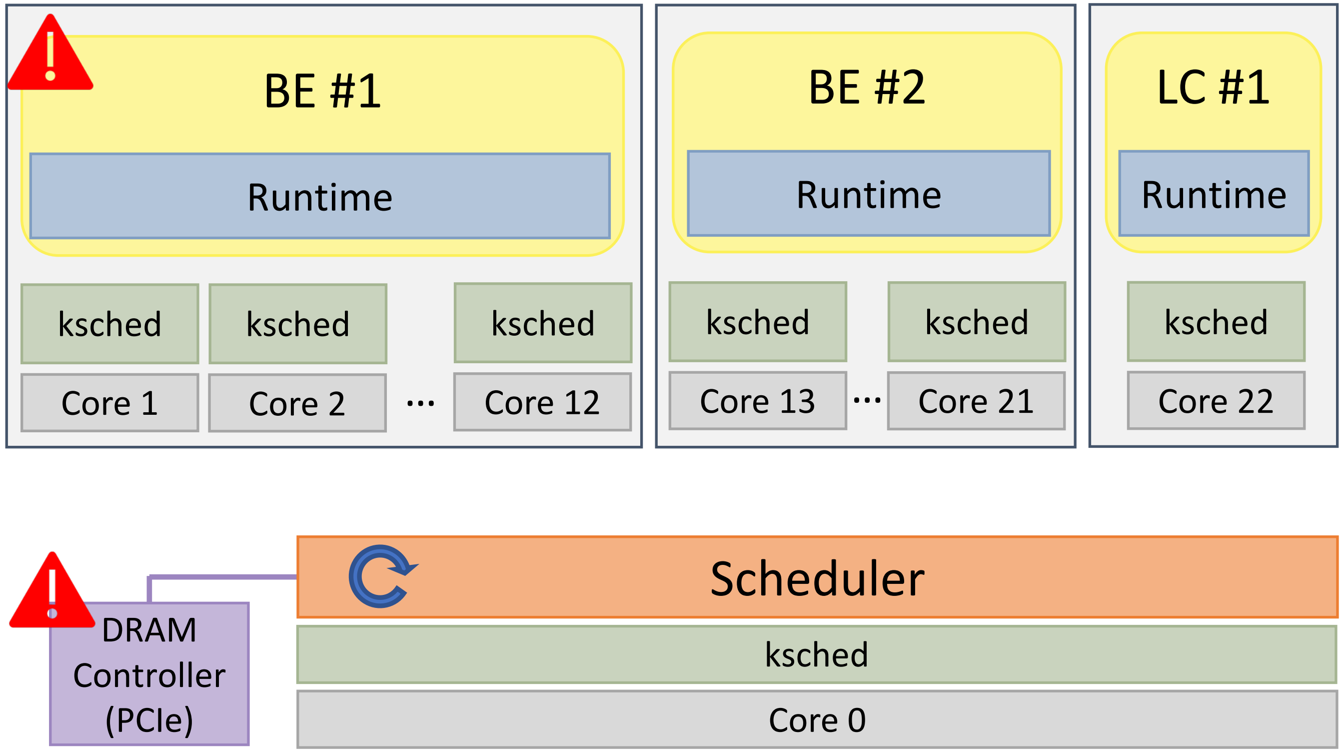


- Policy: keep total bandwidth below target (~80%)
- Detecting Bandwidth Usage:
  - DRAM controller counters
- Identifying an antagonist:
  - per-core LLC miss counters

Example: Mitigating Memory Bandwidth Interference

0 µs:

- BE task 1 begins consuming 100% of memory bandwidth



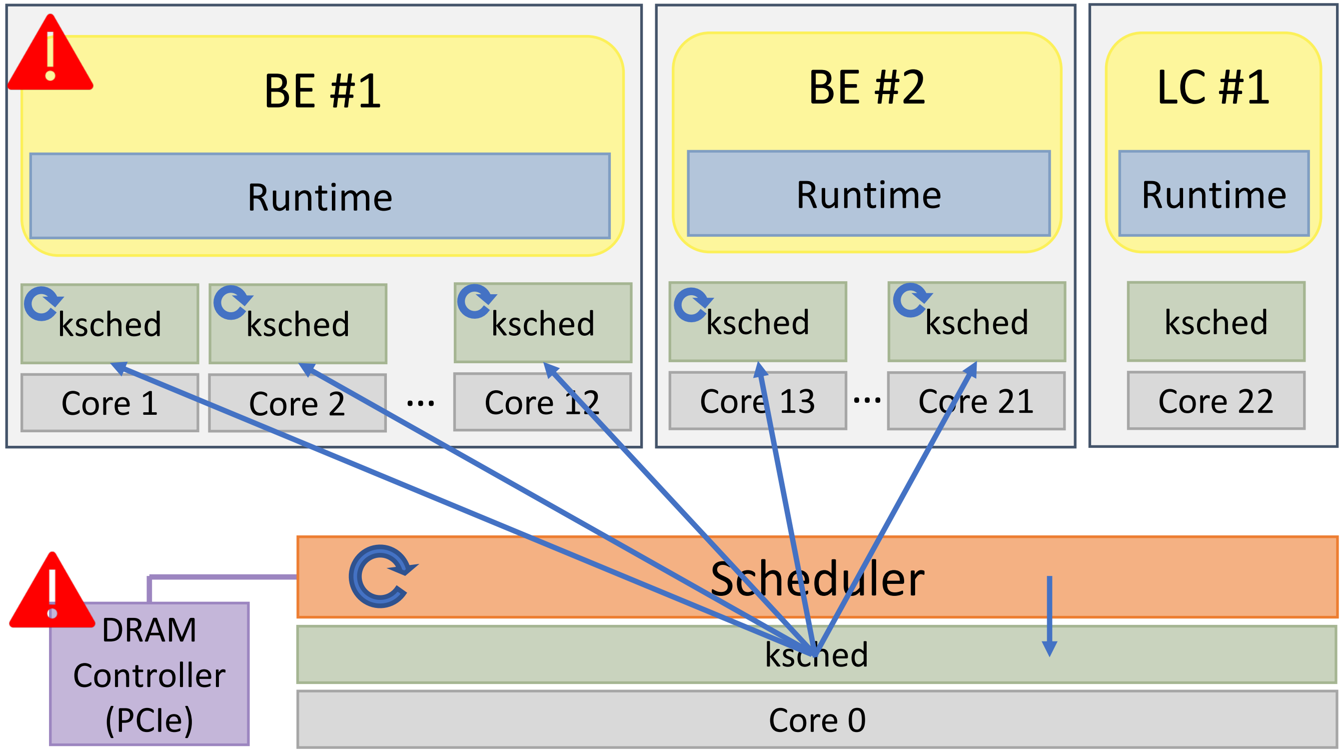
Example: Mitigating Memory Bandwidth Interference

0 µs:

- BE task 1 begins consuming 100% of memory bandwidth

10 µs:

- Interference Detected
- LLC Sampled



Example: Mitigating Memory Bandwidth Interference

0 µs:

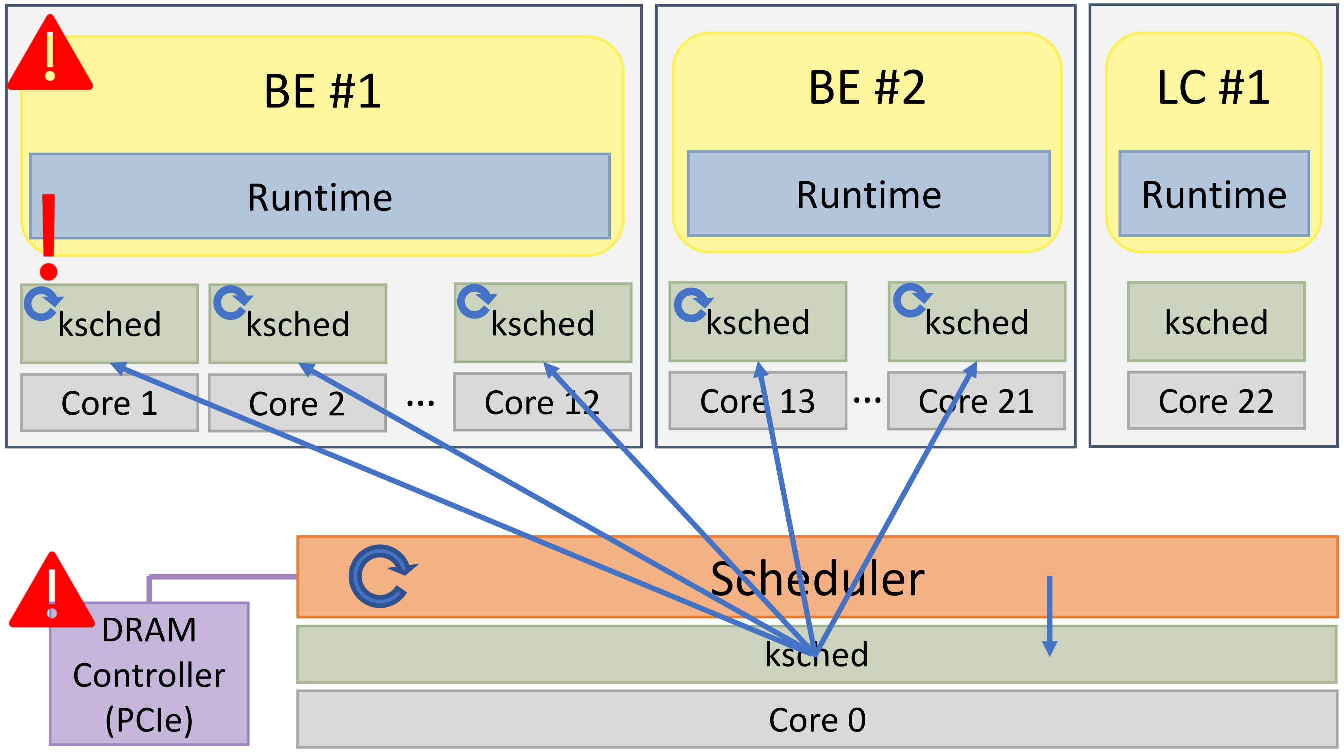
- BE task 1 begins consuming 100% of memory bandwidth

10 µs:

- Interference Detected
- LLC Sampled

20 µs:

- LLC Sampled



Example: Mitigating Memory Bandwidth Interference

0 µs:

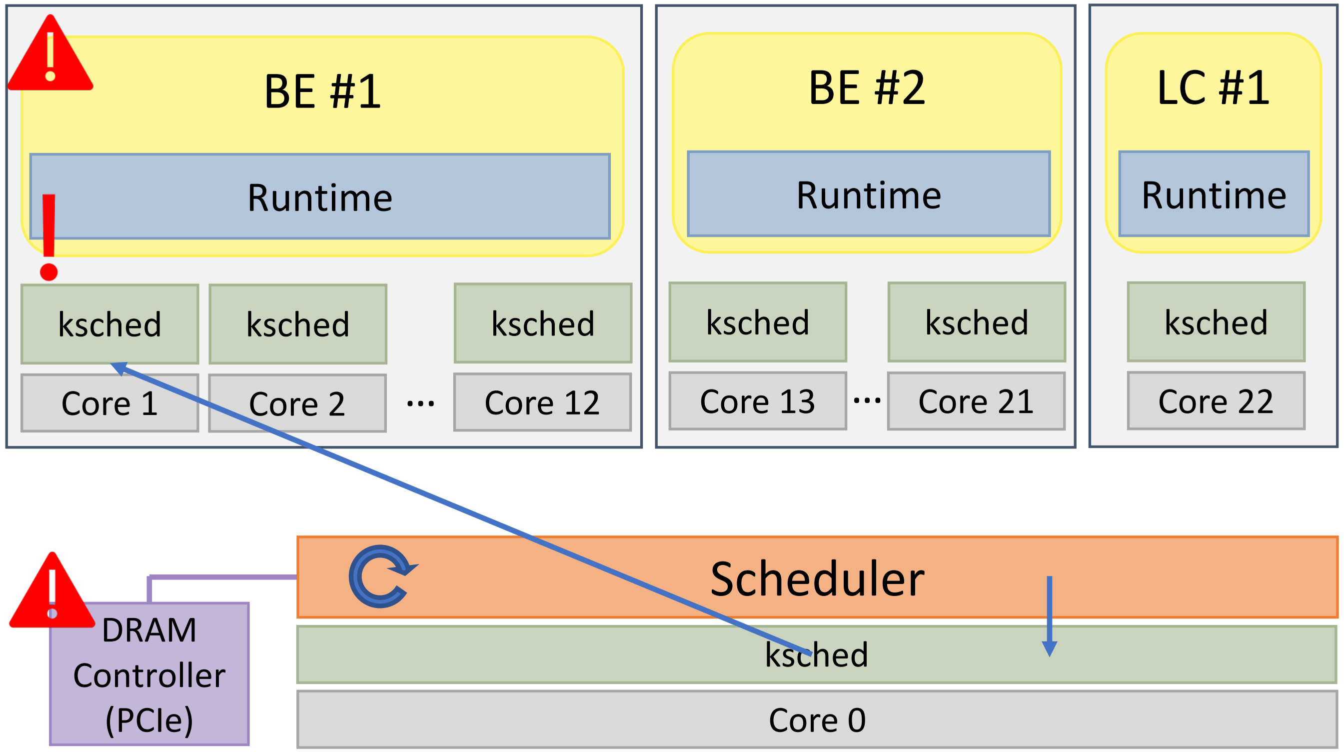
- BE task 1 begins consuming 100% of memory bandwidth

10 µs:

- Interference Detected
- LLC Sampled

20 µs:

- LLC Sampled



Example: Mitigating Memory Bandwidth Interference

0 µs:

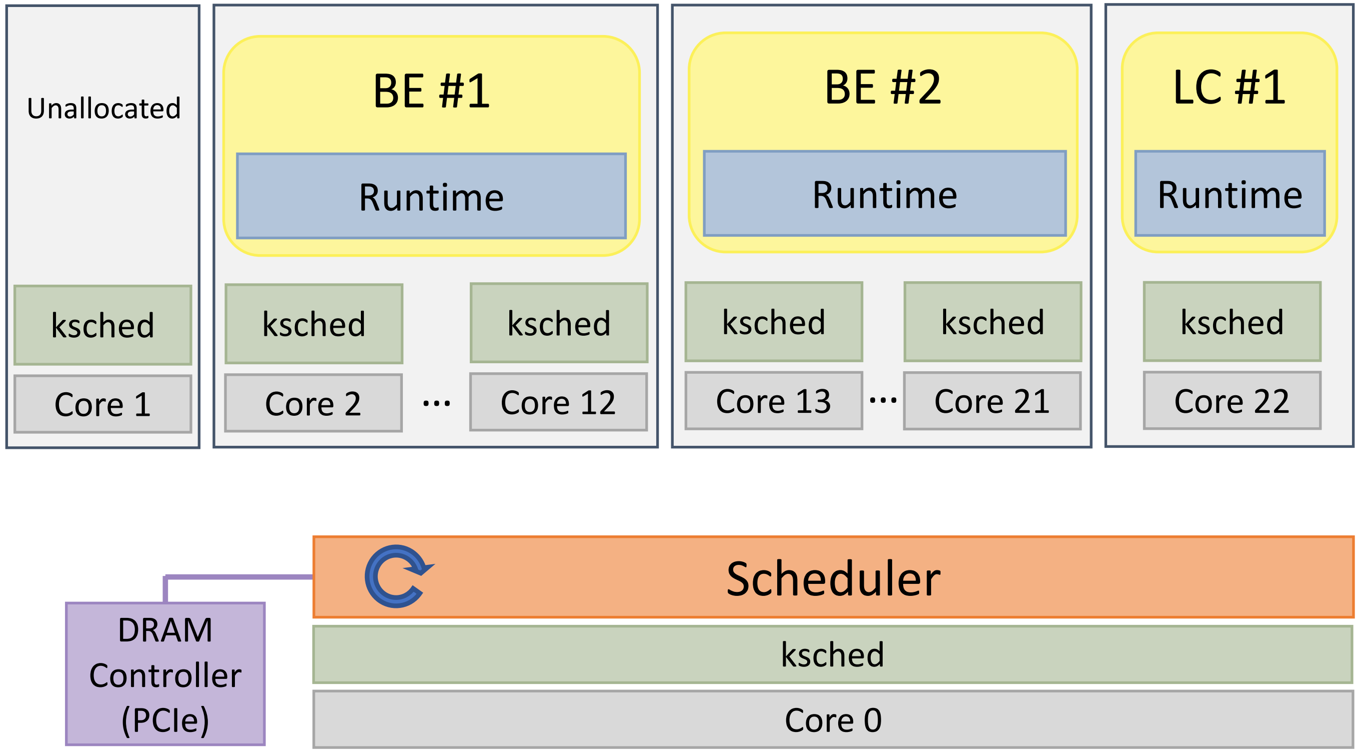
- BE task 1 begins consuming 100% of memory bandwidth

10 µs:

- Interference Detected
- LLC Sampled

20 µs:

- LLC Sampled
- Core revoked from BE #1



Implementation

Scheduler

- Optimized to run the full control loop every 10 µs
- 3500 LOC

KSCHED

- Runs on the Linux Kernel 5.2.0
- Leverages hardware multicast IPIs
- 530 LOC

Runtime

- derived from Shenango [NSDI '19]
- Integrated libibverbs and SPDK to provide direct access to I/O devices
  - 3000 LOC
  - Supports Mellanox ConnectX-5

Evaluation

1. How does Caladan compare to state-of-the-art systems?
2. Are Caladan's benefits generalizable to many tasks sharing a server?

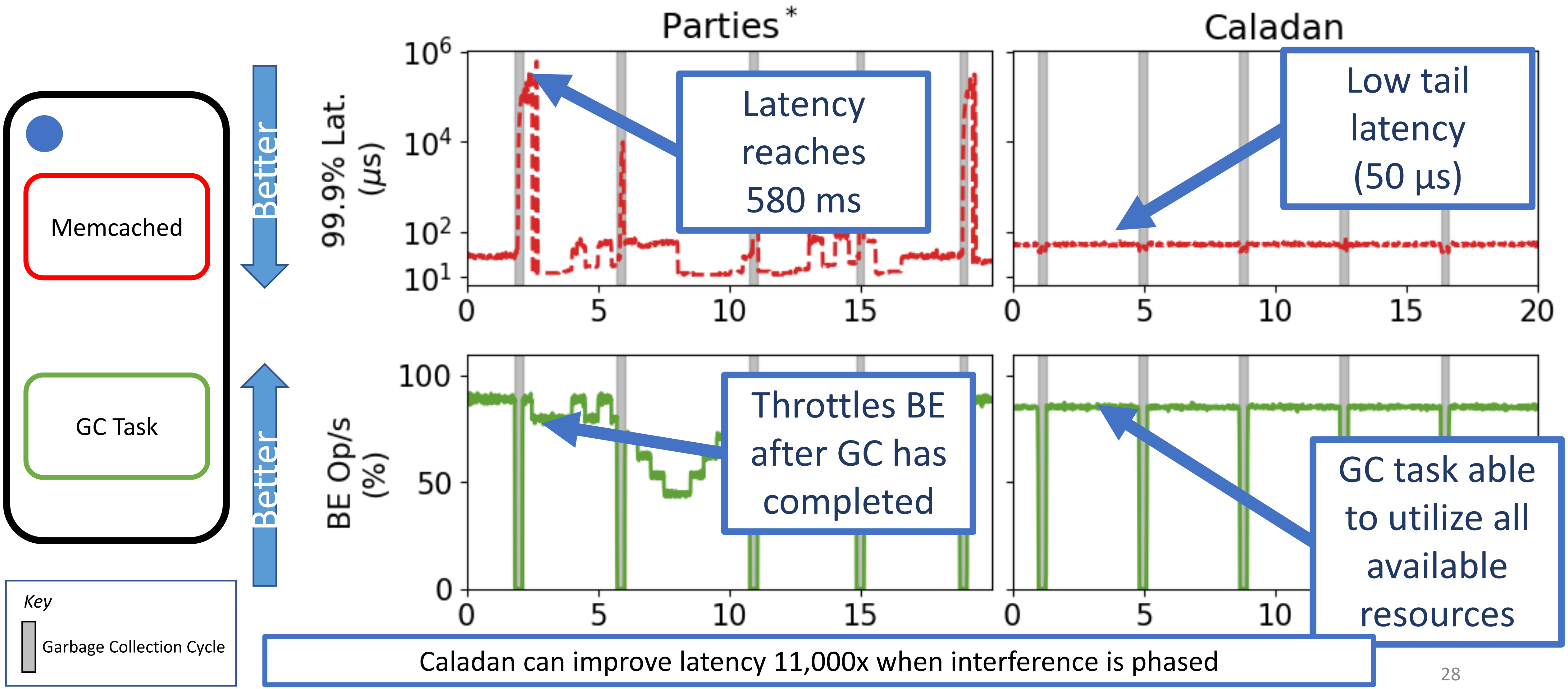
State-of-the-art: Parties [ASPLOS ‘19]

- Adjusts core and cache partitions
- 500 ms decision interval, 10-20 seconds convergence
- *Our implementation: Parties**

Ported Tasks:

- Latency-Critical: memcached, storage service, silo database
- Best-Effort: streamcluster and swaptions-GC (PARSEC)

Memcached and GC



Caladan can improve latency 11,000x when interference is phased

Colocating Many Tasks

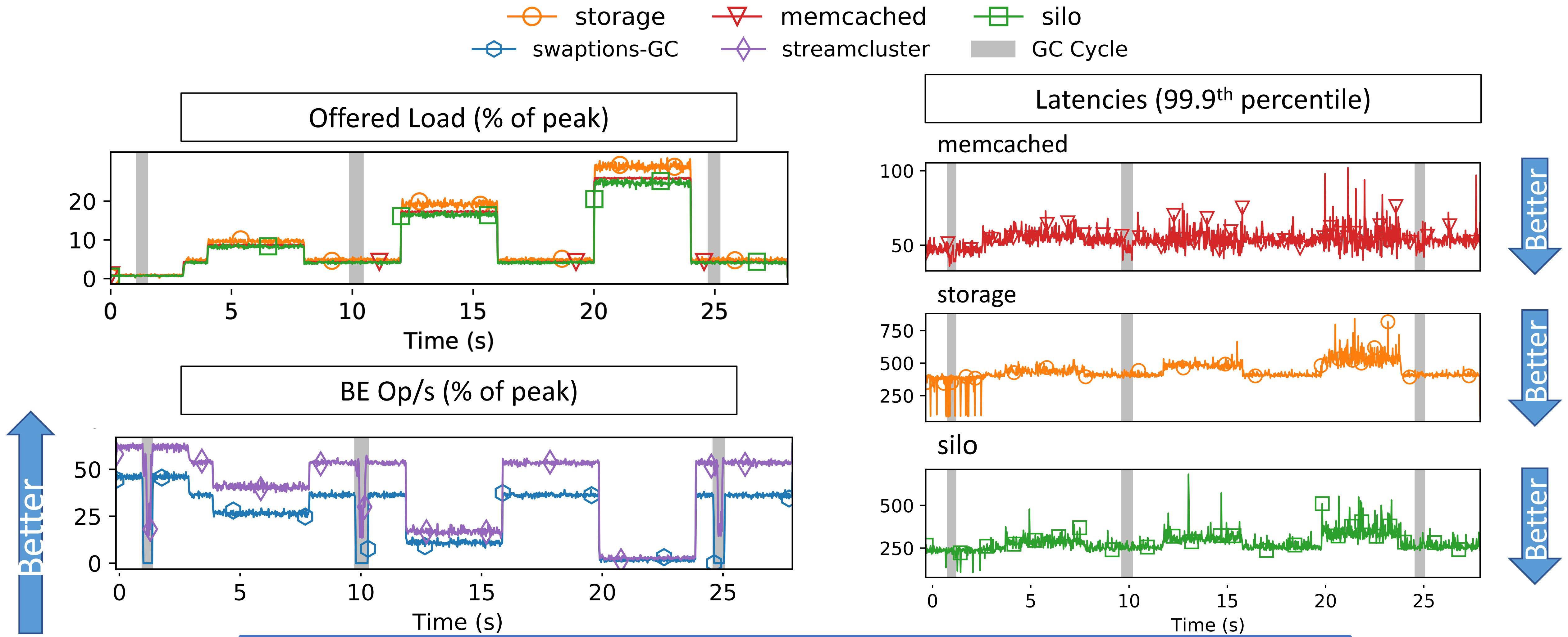
- 3 Latency-Critical Tasks
  - Memcached
  - Flash storage service
  - Silo

- 2 Best-Effort Tasks
  - Swaptions (GC Task)
  - Streamcluster

30 seconds, variable load and interference

Colocating Many Tasks

Core allocations occur up to 230,000 times per second



Caladan maintains low tail latency for all 3 LC tasks under varying load and interference

Requirements for Applications

Applications must link with the runtime

- Export signals, balance work across active cores
- Realistic programming model
  - Partial compatibility layer for some systems libraries

LC applications must expose internal parallelism to runtime

- Example: Memcached modified to spawn a thread per-connection

- Allows scheduler to observe delays
- Allows scheduler to mitigate delays with additional cores

No required changes for BE tasks

Conclusion

Caladan improves machine utilization and performance isolation for low-latency workloads when colocated with noisy best-effort tasks

- Uses no hardware partitioning mechanisms, only rapid core-scheduling
- Uses carefully selected control signals
- Employs many optimizations to make signal collection and core allocation efficient

- Offers 11,000x latency improvement over the state-of-the-art for a latency-critical workload when there is phased interference

https://github.com/shenango/caladan