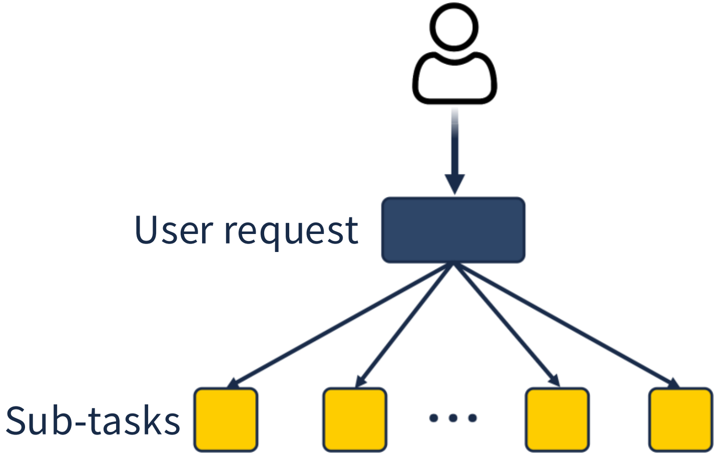
UC San Diego

The Benefits and Limitations of *User Interrupts* for Preemptive Userspace Scheduling

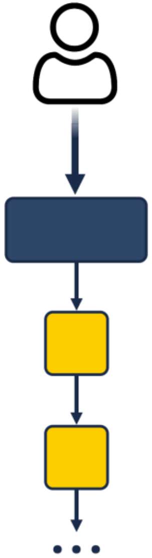
Linsong Guo, Danial Zuberi, Tal Garfinkel, Amy Ousterhout

22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI' 25) | April 29, 2025

Datacenters Need µs-Scale Tail Latencies



High fan-out services[1]



Data-dependent services

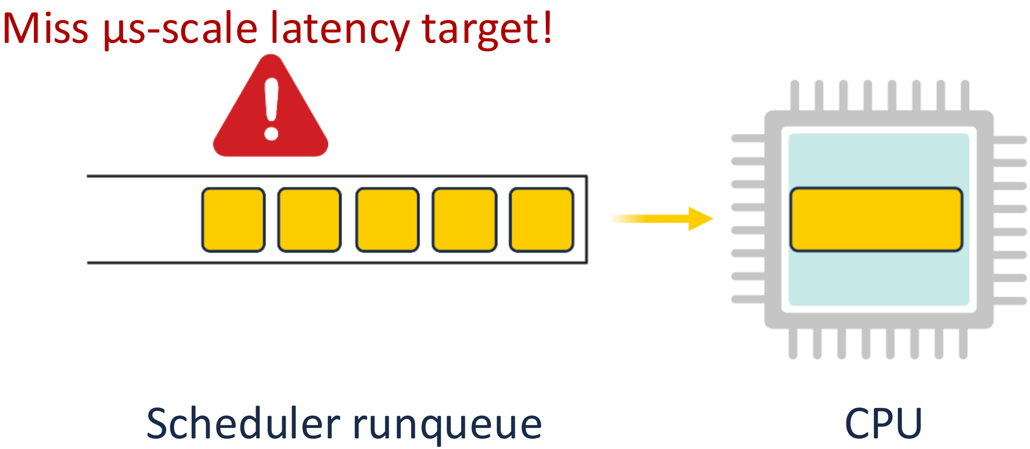
[1] Jeffrey Dean, Luiz André Barroso, The tail at scale.

Service Time Varies

Short-running task
(a few μs)

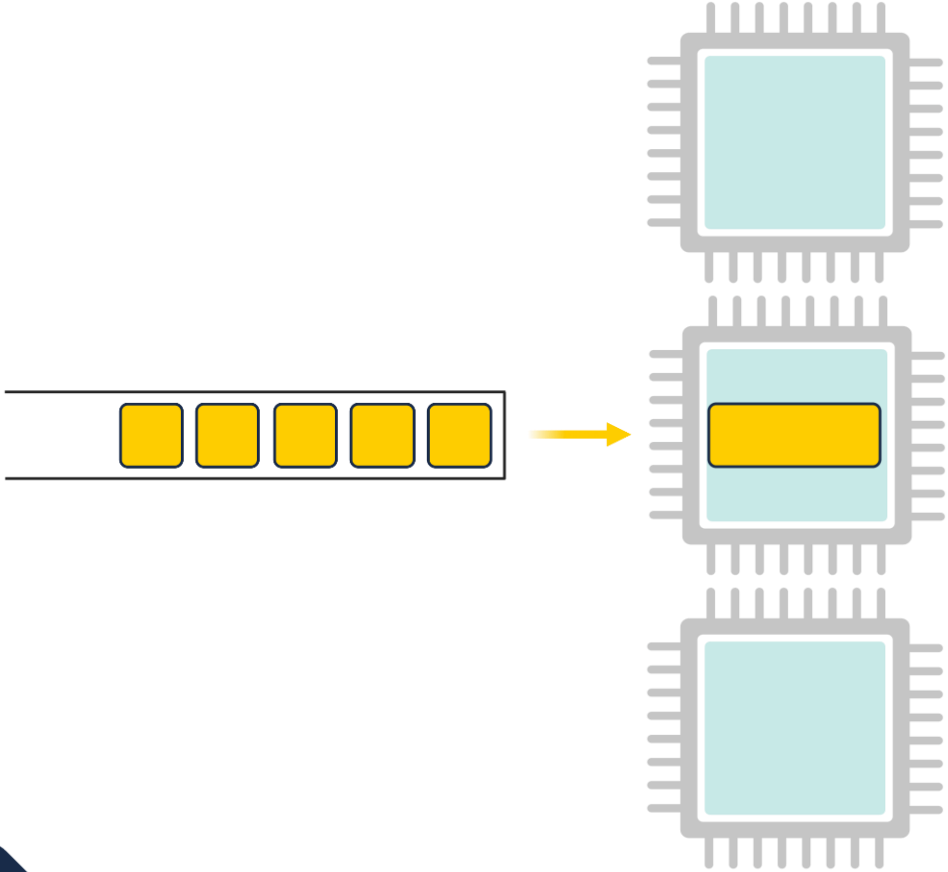
Long-running task
(hundreds of μs, or even ms)

Problem: Head-of-Line Blocking

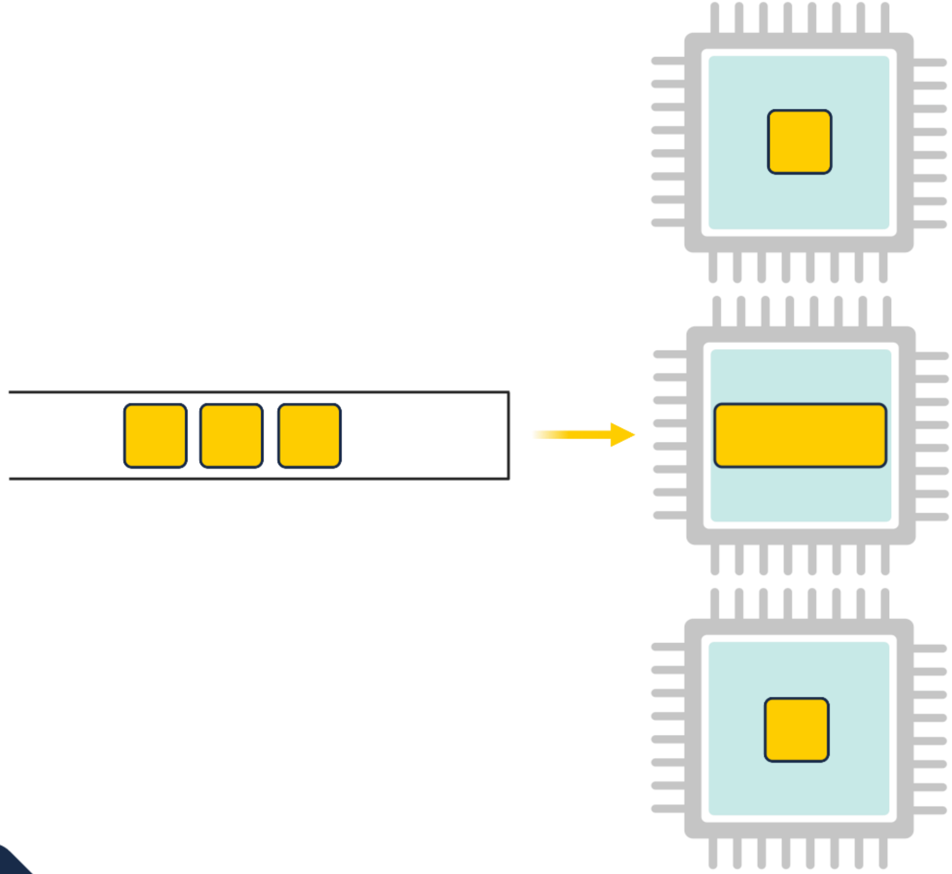


Example: transactional tasks and analytical tasks in databases.

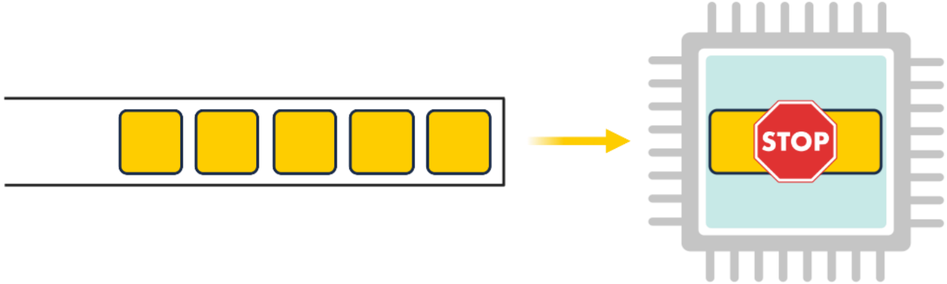
Solution #1: Overprovisioning Wastes CPUs



Solution #1: Overprovisioning Wastes CPUs



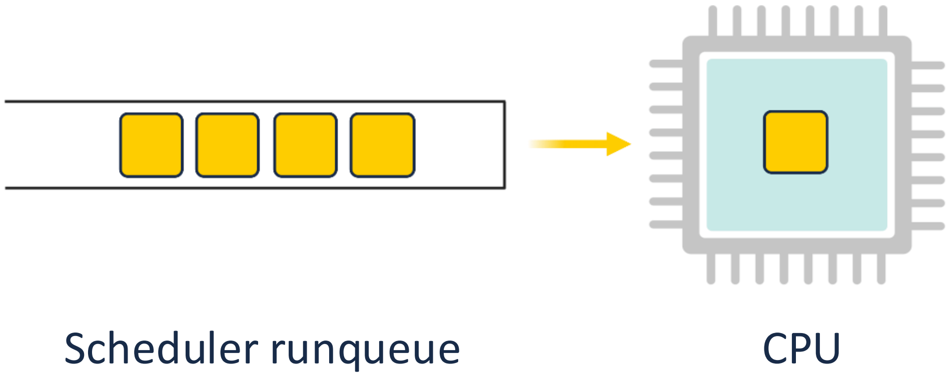
Solution #2: Fine-grained Preemption



Scheduler runqueue

CPU

Solution #2: Fine-grained Preemption



Limitations of User-Space Preemption

UC San Diego

Mechanism #1: Compiler Instrumentation

The compiler instruments **"poll and yield"** code throughout user programs (e.g., at loop back-edges or function calls).

One timer core updates the `shared_var`.

```
if (shared_var == true) {
    Yield();
}
```

Examples: Go, wasmtime, Concord (SOSP' 23)

Mechanism #2: Signals

One timer core sends **signals** to preempt running threads.

Receiving signals is expensive because this involves **kernel space.**

Examples: Go

New Opportunity: User Interrupts

UC San Diego

- A hardware technique that sends and receives interrupts in user space.
- Available in Intel’s CPUs since 2023.
- Lower receiving overhead than signals (~0.4 µs vs. 2.4 µs).

Contributions

UC San Diego

? **Can user interrupts help achieve fine-grained preemption?**

1 Microbenchmark study — basic overhead of preemption mechanisms

2 Developed two preemptive user-space runtimes:

- Aspen-KB (kernel-bypass runtime)
- Aspen-Go (extended Go runtime)

3 Application study — overall performance of preemption mechanisms

1 Microbenchmark Study

Experiment Setup: Preempt benchmark program with three preemption mechanisms:

(1) Signals

(2) User interrupts

(3) Compiler instrumentation (implemented with Concord[1]).

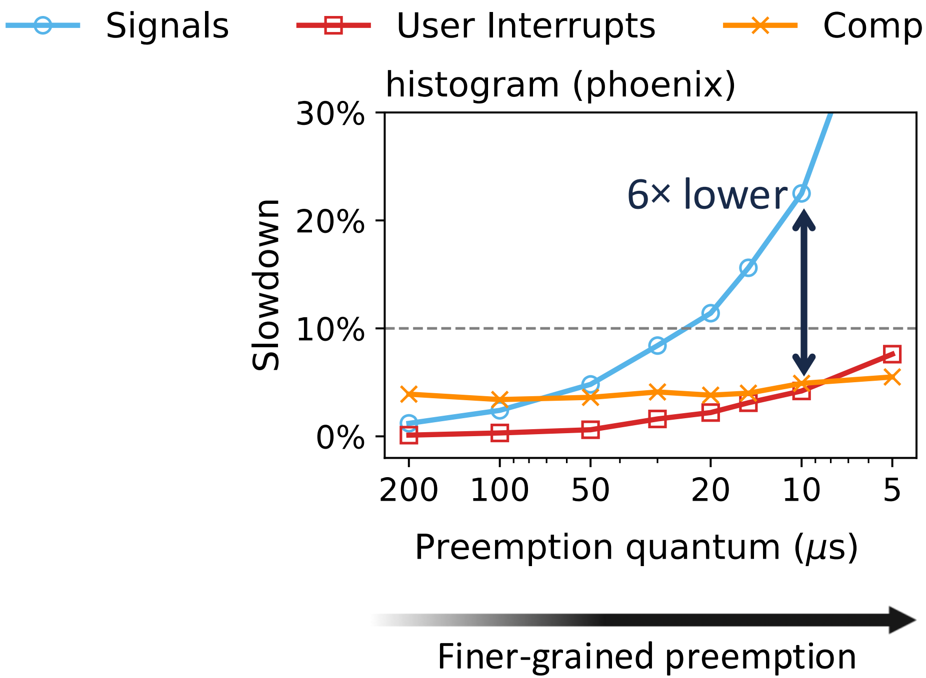
Benchmark Suites: Splash-2, Phoenix, and Parsec.

Metric: Runtime slowdown relative to non-preemptive execution.

[1] Rishabh Iyer et al., *Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling*, SOSP, 2023.

Signals vs. User Interrupts

One representative benchmark program: *histogram*

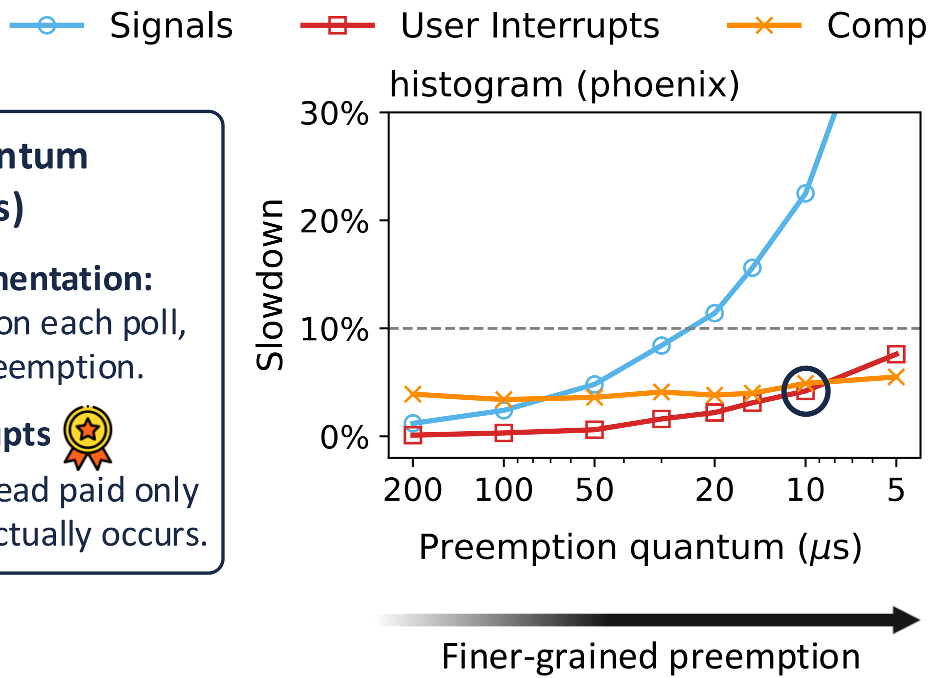


User interrupts have significantly lower overhead than signals.

User Interrupts vs. Compiler Instrumentation

UC San Diego

One representative benchmark program: *histogram*



Larger quantum
(> 10 μs)

Compiler Instrumentation:
CPU cycles wasted on each poll, even without preemption.

User Interrupts

No polling — overhead paid only when preemption actually occurs.

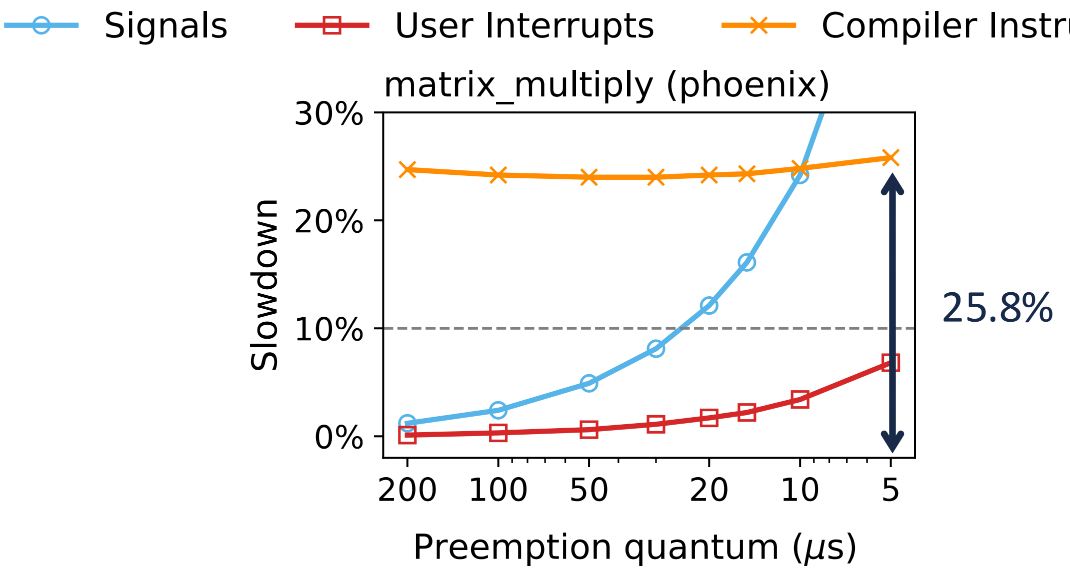
Smaller quantum
(< 10 μs)

Compiler Instrumentation

Polling becomes efficient as preemption is more likely to occur with smaller quantum.

Compiler Instrumentation: Variable Across Programs

Example: Programs with tight loops may incur unpredictably high overhead.



Configuration Challenges

Where to instrument? At loops? Unroll loops? At function calls? Different program inputs?

Tradeoffs Between Preemption Mechanisms

UC San Diego

Signals	Compiler Instrumentation	User Interrupts
High Overhead with µs-scale quantum.	Lower overhead with **smaller** quantum; Unpredictably **high** overhead in some programs; Challenging to configure.	Lower overhead with **larger** quantum; **Consistent** overhead; No configure required.

UC San Diego

2 Preemptive User-level Schedulers

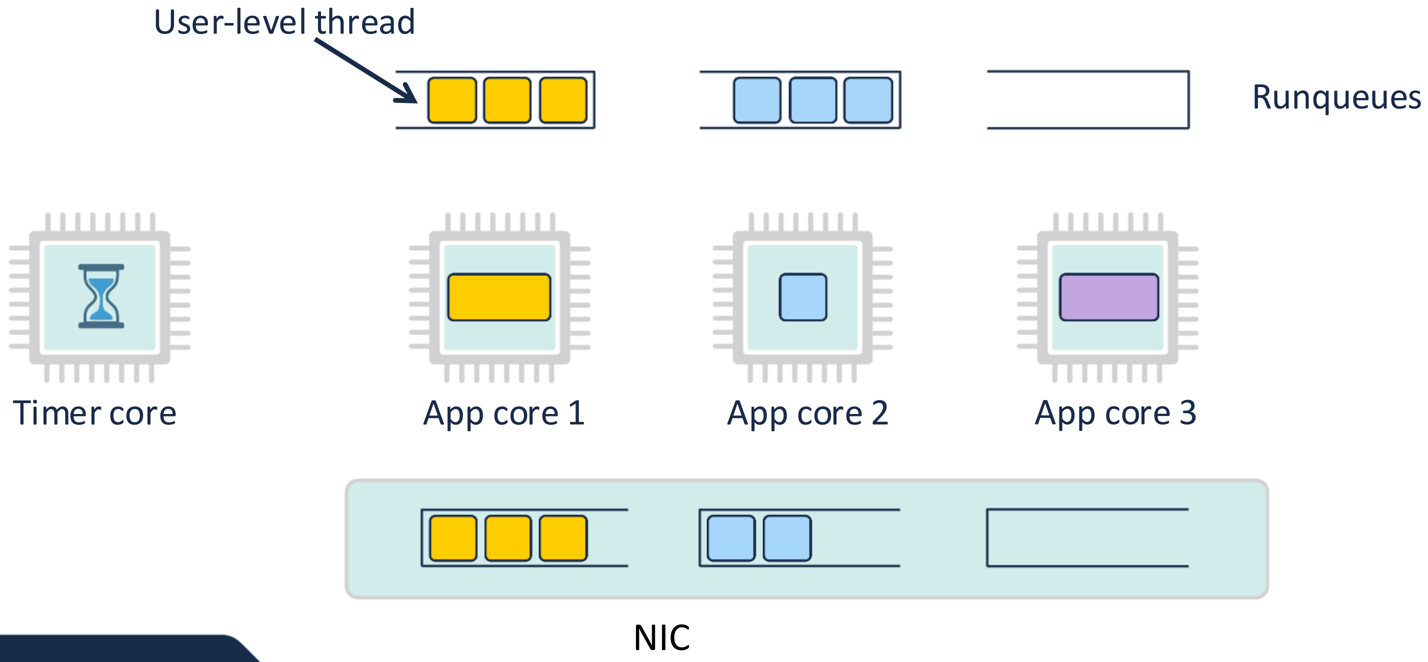
Built two preemptive user-level schedulers with user interrupts:

- ***Aspen-KB*** is built on a kernel-bypass runtime, ***Caladan***[1].
- ***Aspen-Go*** extends the popular ***Go*** runtime.

[1] Joshua Fried et al., *Caladan: Mitigating Interference at Microsecond Timescales*, USENIX NSDI, 2020.

Aspen-KB Design

Common design: A dedicated timer core handles timing and preempts app cores.

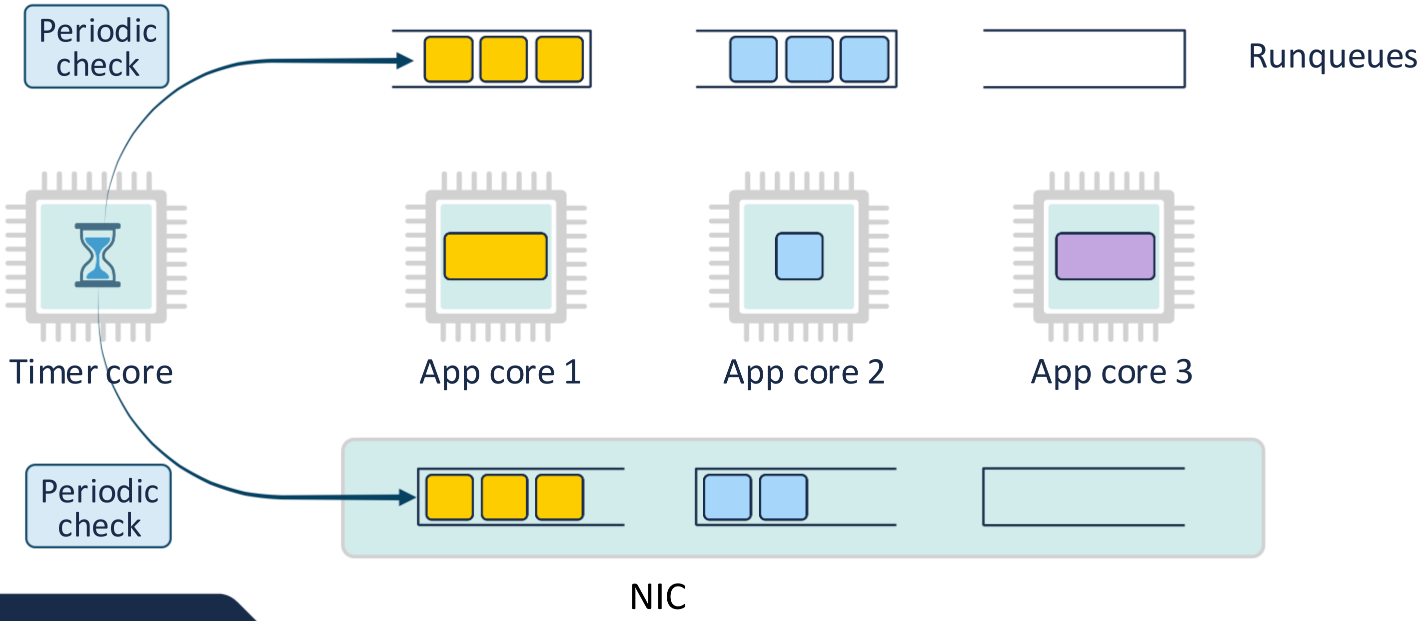


Aspen-KB Design

Existing schedulers: Preempt app cores periodically → high preemption cost.

Policy #1: Preempt only when necessary.

Preempt only when NIC receive queue or scheduler runqueue has waiting tasks.



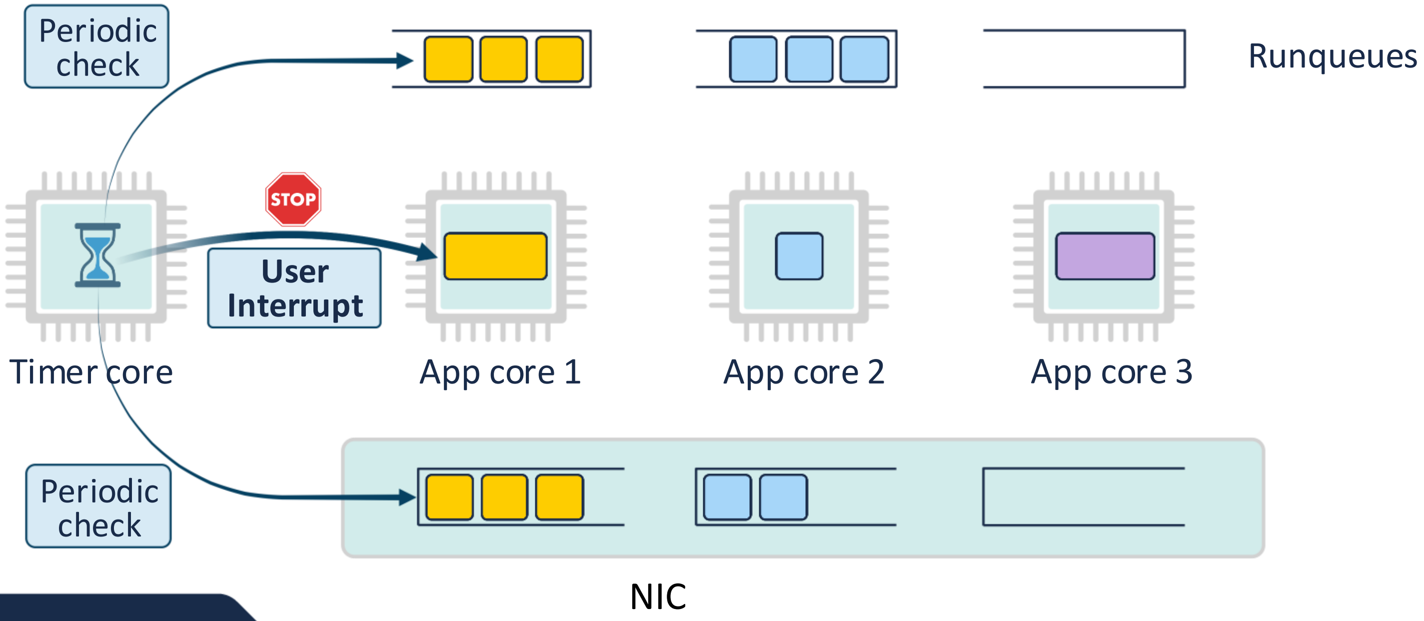
Aspen-KB Design

UC San Diego

Existing schedulers: Preempt app cores periodically → high preemption cost.

Policy #1: Preempt only when necessary.

Preempt only when NIC receive queue or scheduler runqueue has waiting tasks.

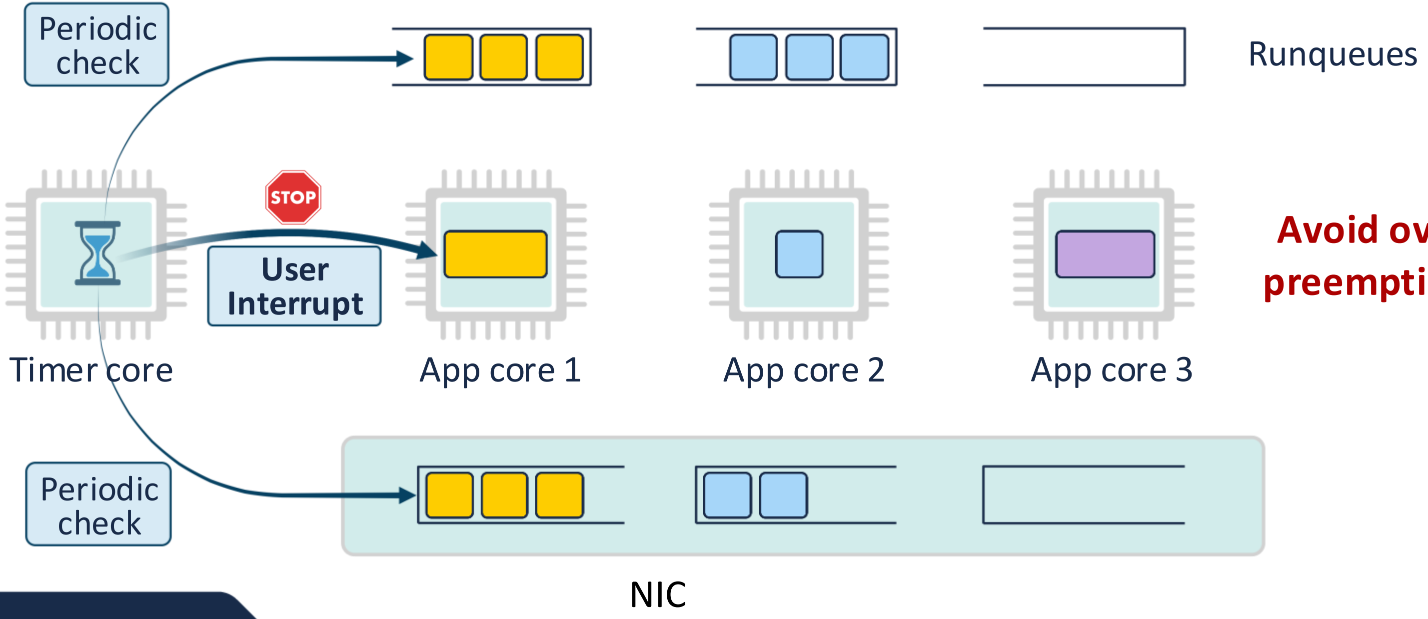


Aspen-KB Design

Existing schedulers: Preempt app cores periodically → high preemption cost.

Policy #1: Preempt only when necessary.

Preempt only when NIC receive queue or scheduler runqueue has waiting tasks.

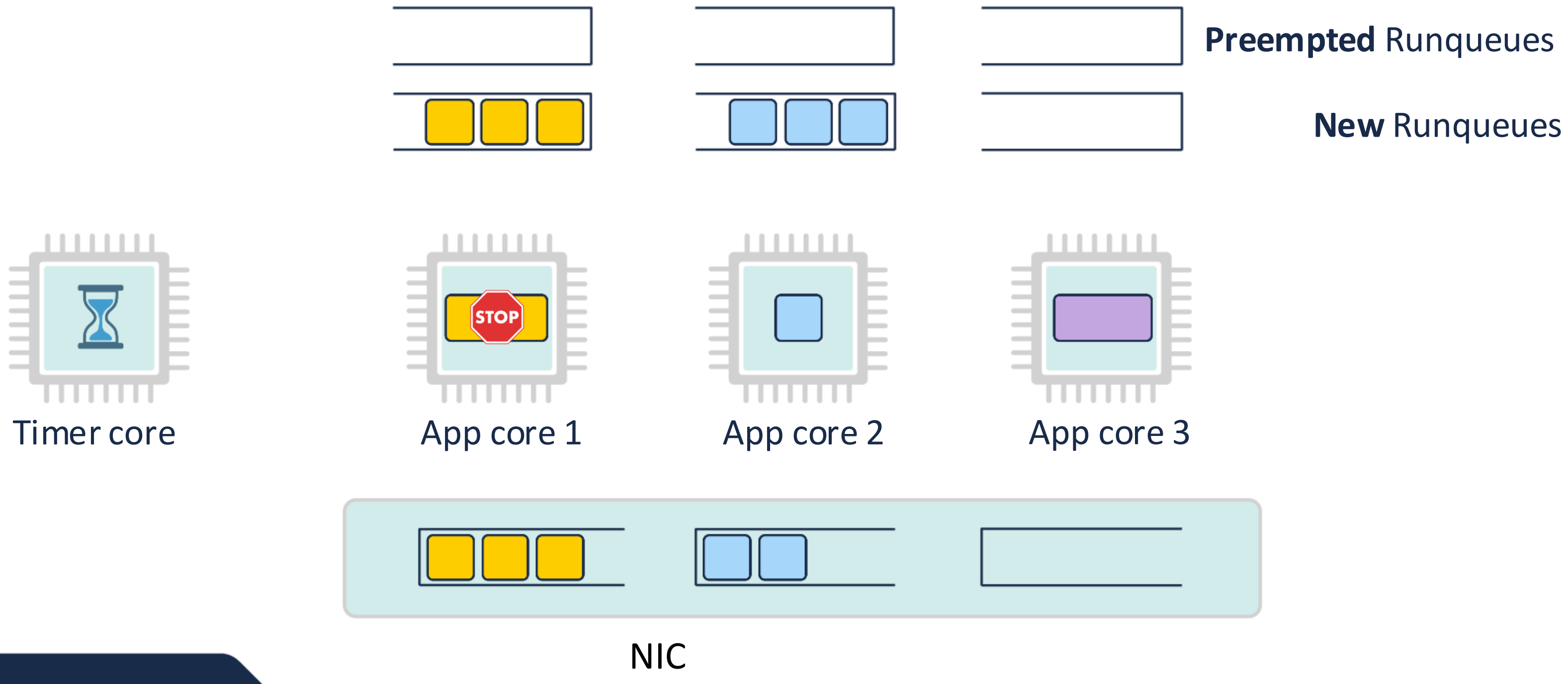


Avoid over 70% of preemption events!

Aspen-KB Design

Policy #2: Two-queue scheduling policy.

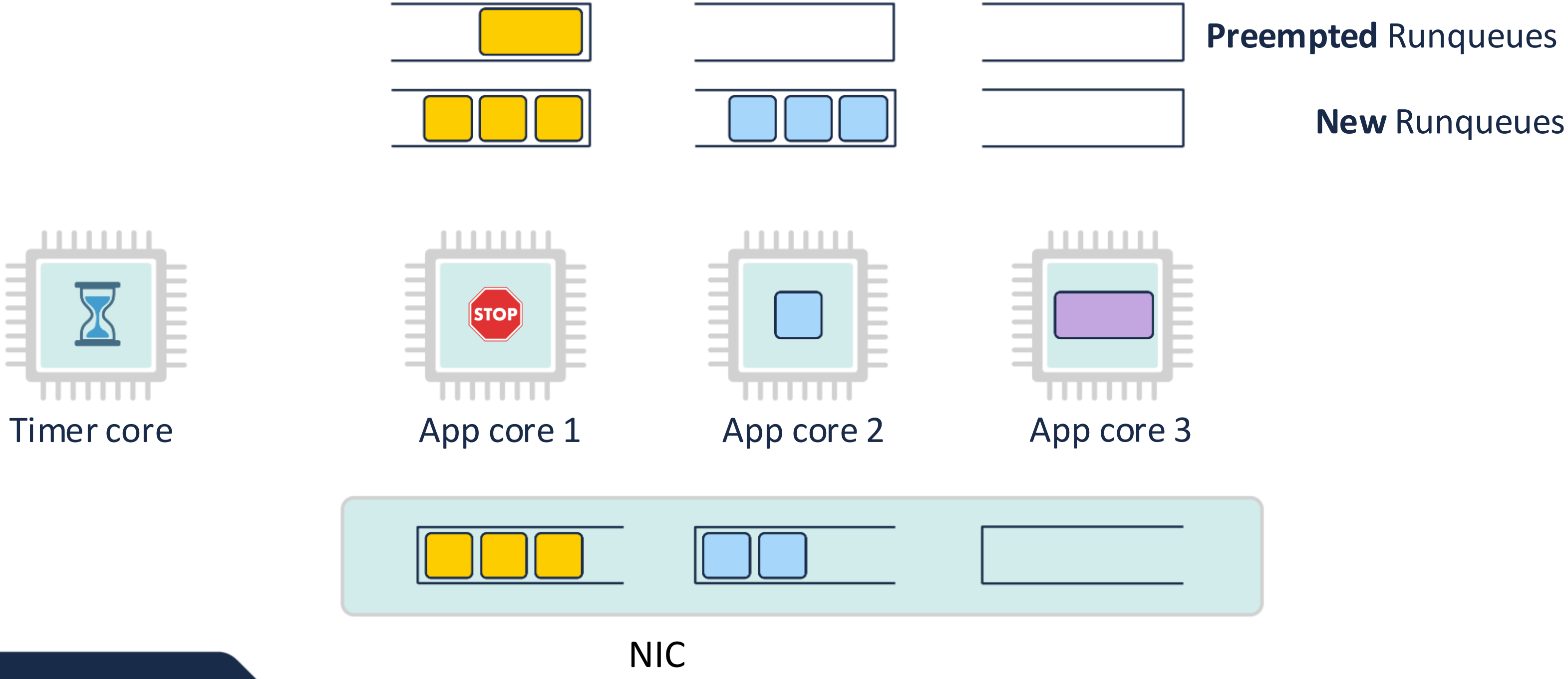
Prioritizes tasks from the new queue over the preempted queue.



Aspen-KB Design

Policy #2: Two-queue scheduling policy.

Prioritizes tasks from the new queue over the preempted queue.

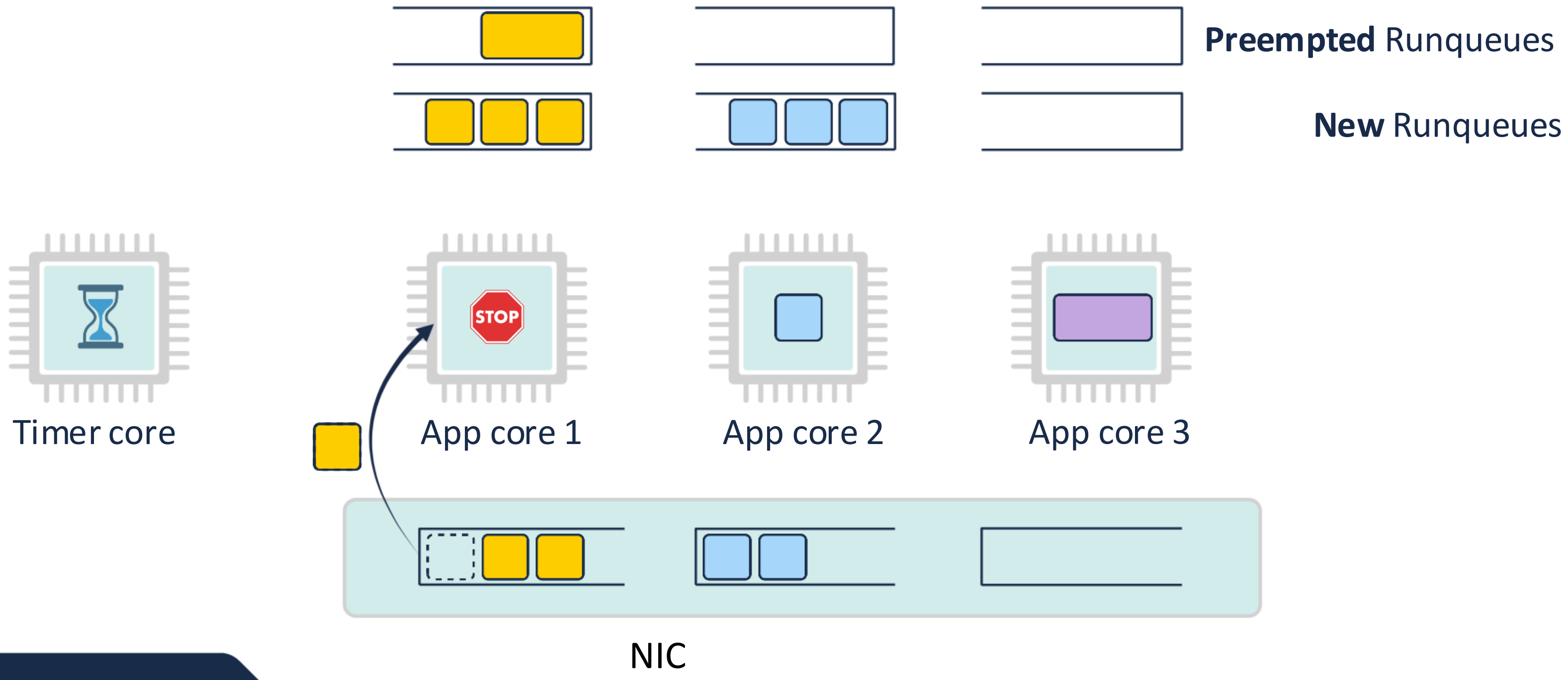


Aspen-KB Design

Existing schedulers: Infrequent packet polling → new packets blocked in network stack.

Policy #3: Match polling and preemption frequencies.

App cores poll network stack at every preemption.

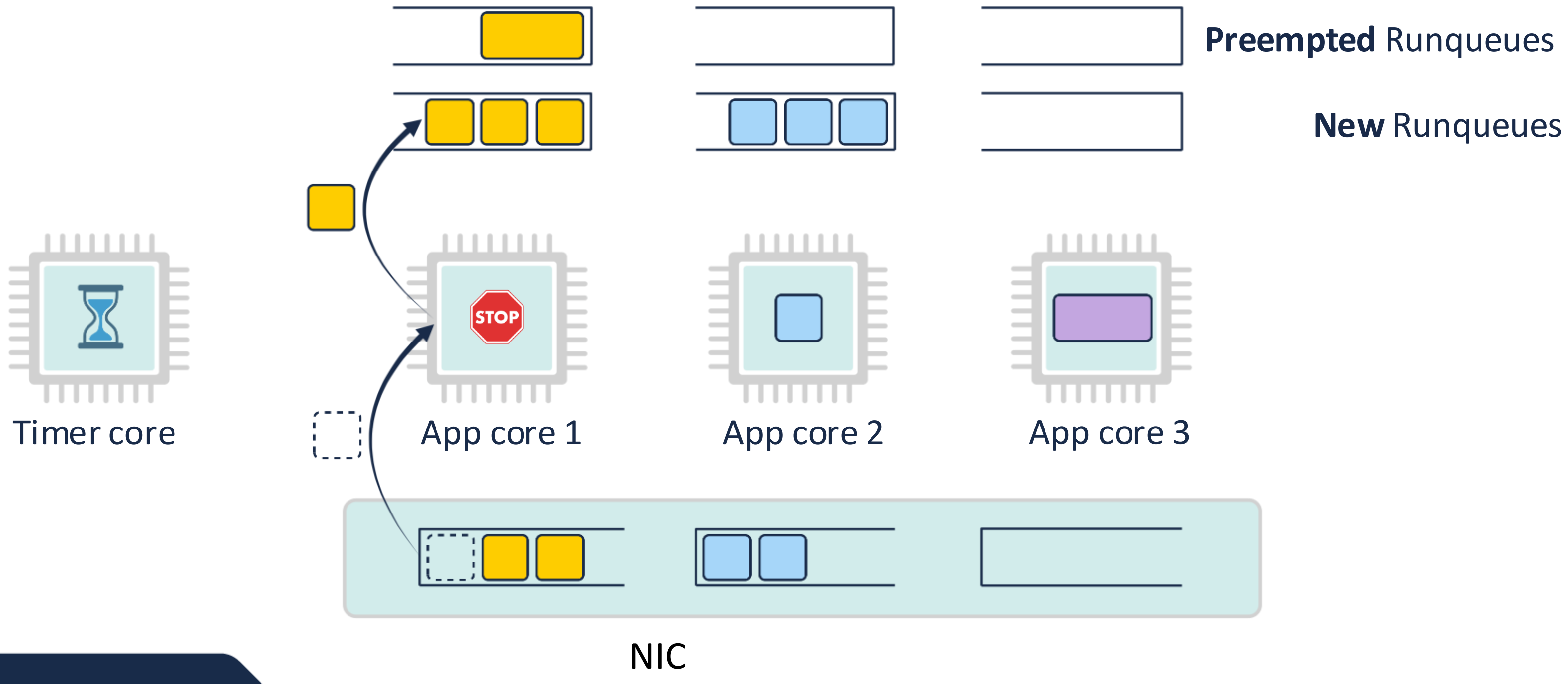


Aspen-KB Design

Existing schedulers: Infrequent packet polling → new packets blocked in network stack.

Policy #3: Match polling and preemption frequencies.

App cores poll network stack at every preemption.



Limitations of *Aspen-Go*

UC San Diego

Aspen-KB

Match network polling and preemption frequencies.

Aspen-Go

(Makes minimal changes in Go)

Go:

Only poll when scheduler runqueue is empty.

Aspen-Go:

Offloads frequent polling to a timer core that polls every 100 µs.

Aspen-Go is weaker than *Aspen-KB* at preventing **head-of-line blocking**.

Limitations of *Aspen-Go*

UC San Diego

Aspen-KB

Preempt only when necessary

Low context-switch overhead of 0.2–0.9 µs

Aspen-Go

Preempt periodically

Complicated scheduler logic with context-switch overhead of 1.3–3.0 µs

Aspen-Go has a high preemption cost.

3 Application Performance Evaluation

- 1 server + 1 client
- Client: runs load generator
- Server: runs applications on Aspen to compare different preemption mechanisms:

  (1) Signals

  (2) User interrupts

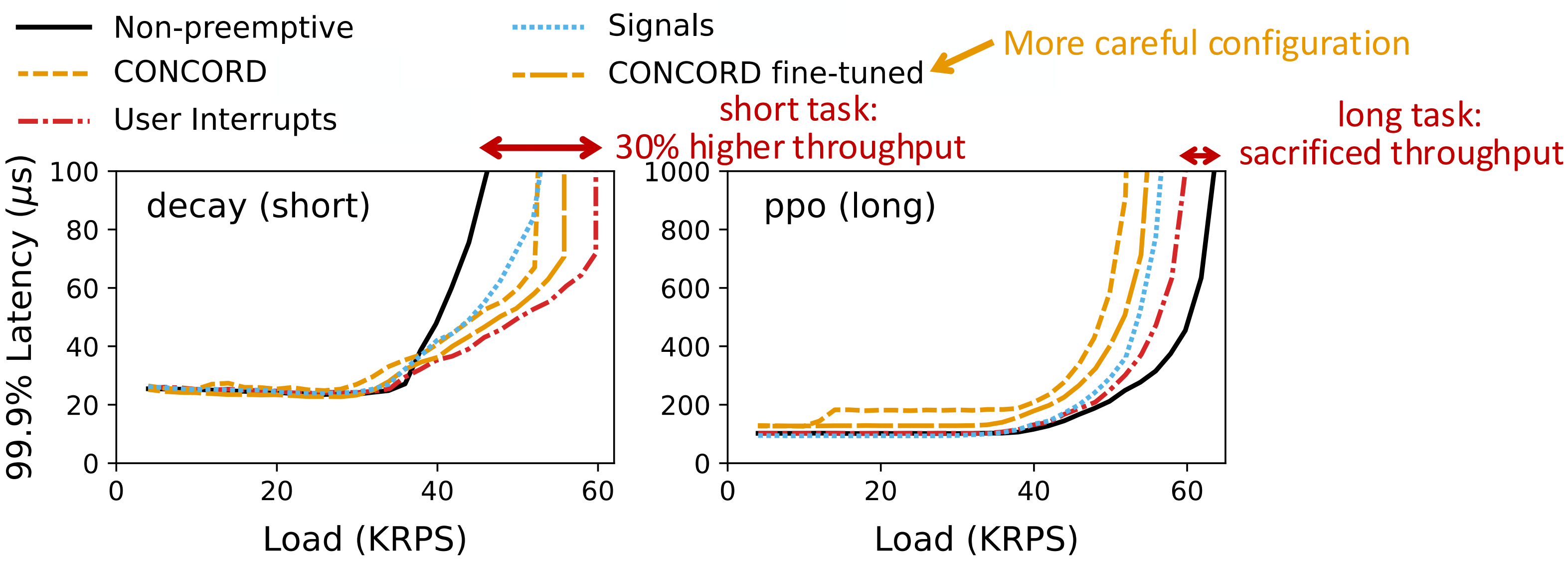
  (3) Compiler Instrumentation (implemented with Concord[1])

[1] Rishabh Iyer et al., *Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling*, SOSP, 2023.

Aspen-KB — DataFrames

UC San Diego

Workload: decay (5 µs), ad (7 µs), rmv (28 µs), ppo (75 µs), kmeans (250 µs); 20% each

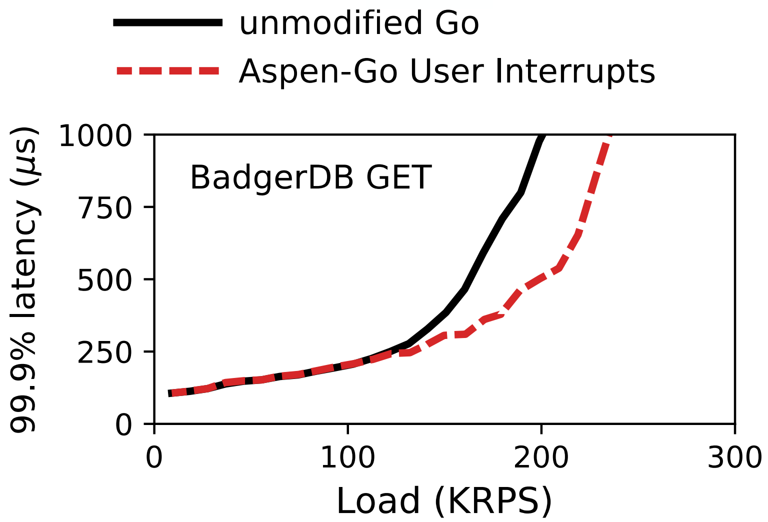


Conclusion: **Aspen-KB** with user interrupts can reduce head-of-line blocking in µs-scale workloads.

Aspen-Go — BadgerDB

UC San Diego

Workload: 99% GET task (5 µs) and 1% SCAN task (800 µs)



Conclusion: Aspen-Go provides limited performance gains with minimal changes to Go.

Conclusion

UC San Diego

Can user interrupts help achieve fine-grained preemption?

Yes, user interrupts can help.

But when a system is not fully optimized for fine-grained preemption, user interrupts provide limited benefits.