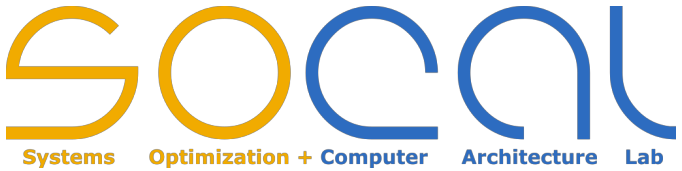
µDPM: Dynamic Power Management for the Microsecond Era

Chih-Hsun Chou
cchou001@cs.ucr.edu

Laxmi N. Bhuyan
bhuyan@cs.ucr.edu

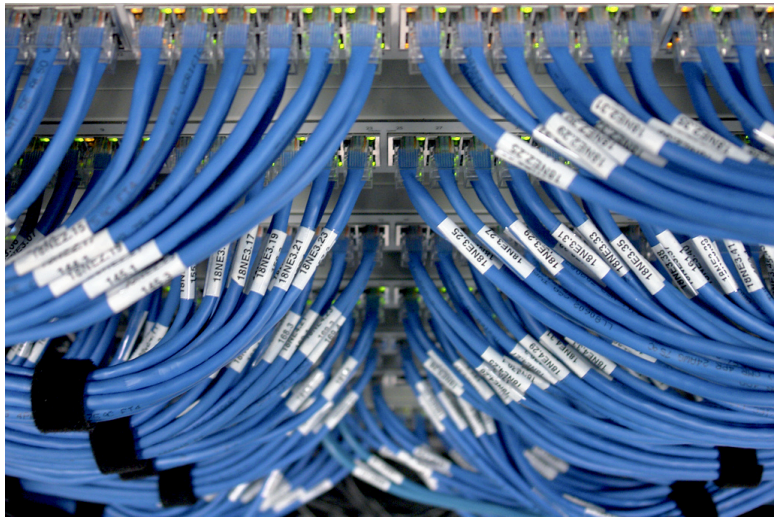
Daniel Wong
danwong@ucr.edu

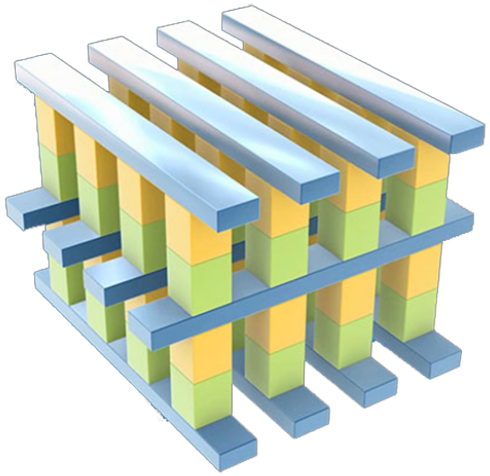


Computer systems efficiently support . . .

ns μs ms events

Masked by microarchitectural techniques





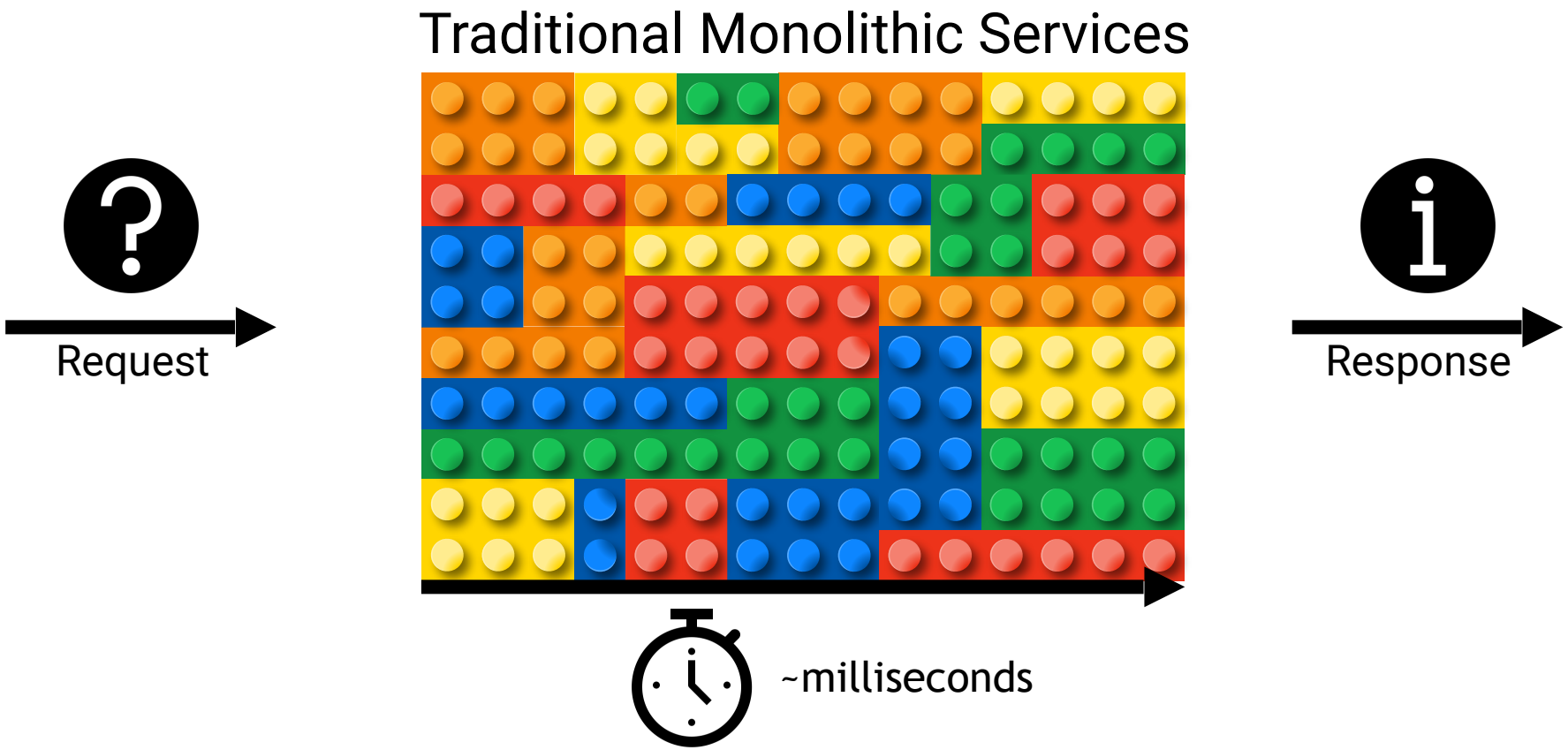
Masked by OS-level techniques

The Killer Microseconds

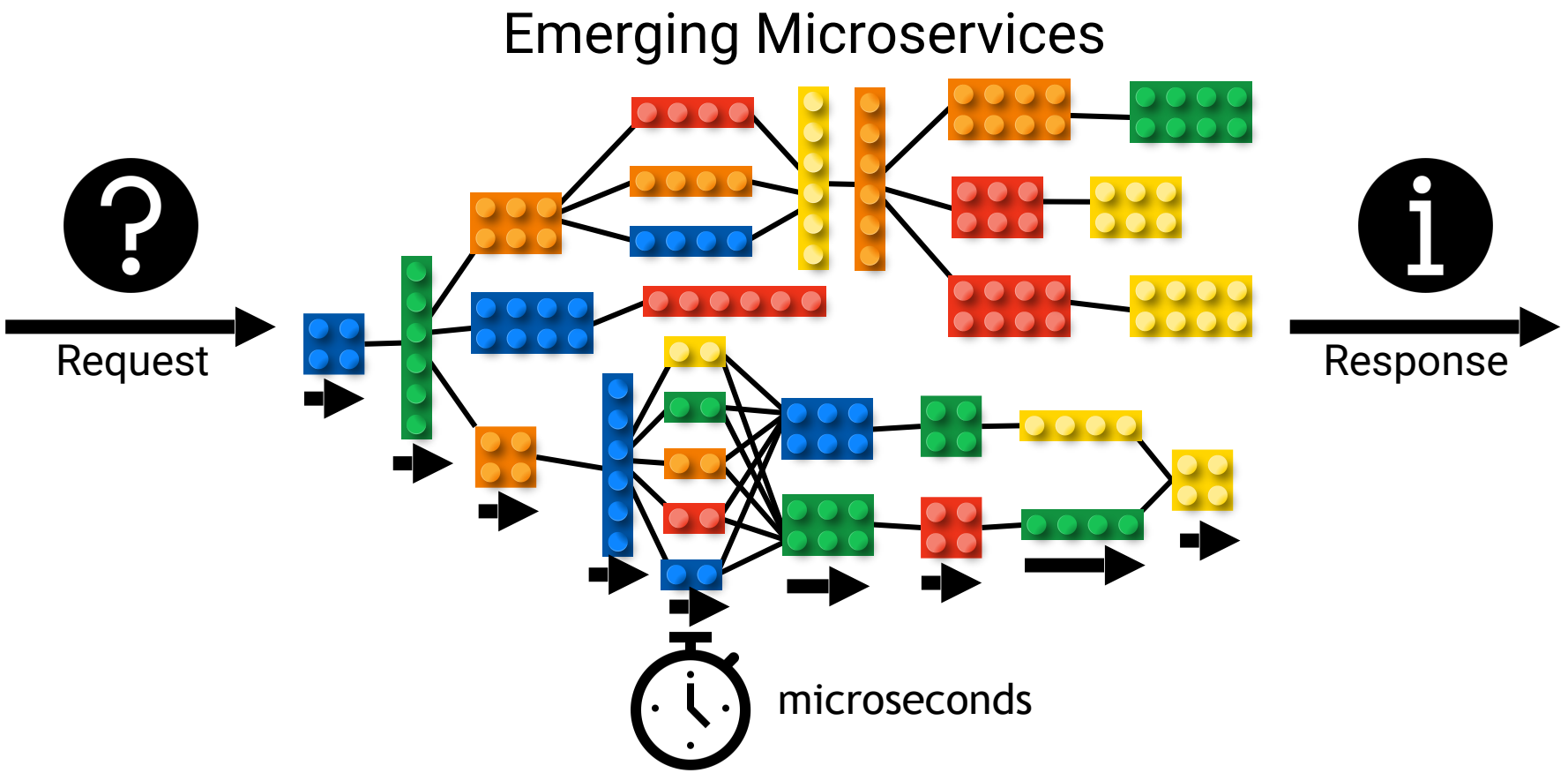
"System designers can no longer ignore efficient support for microsecond-scale I/O ... *Novel microsecond-optimized system stacks are needed*" [1]

[1] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. Commun. ACM 60, 4 (March 2017), 48-54.

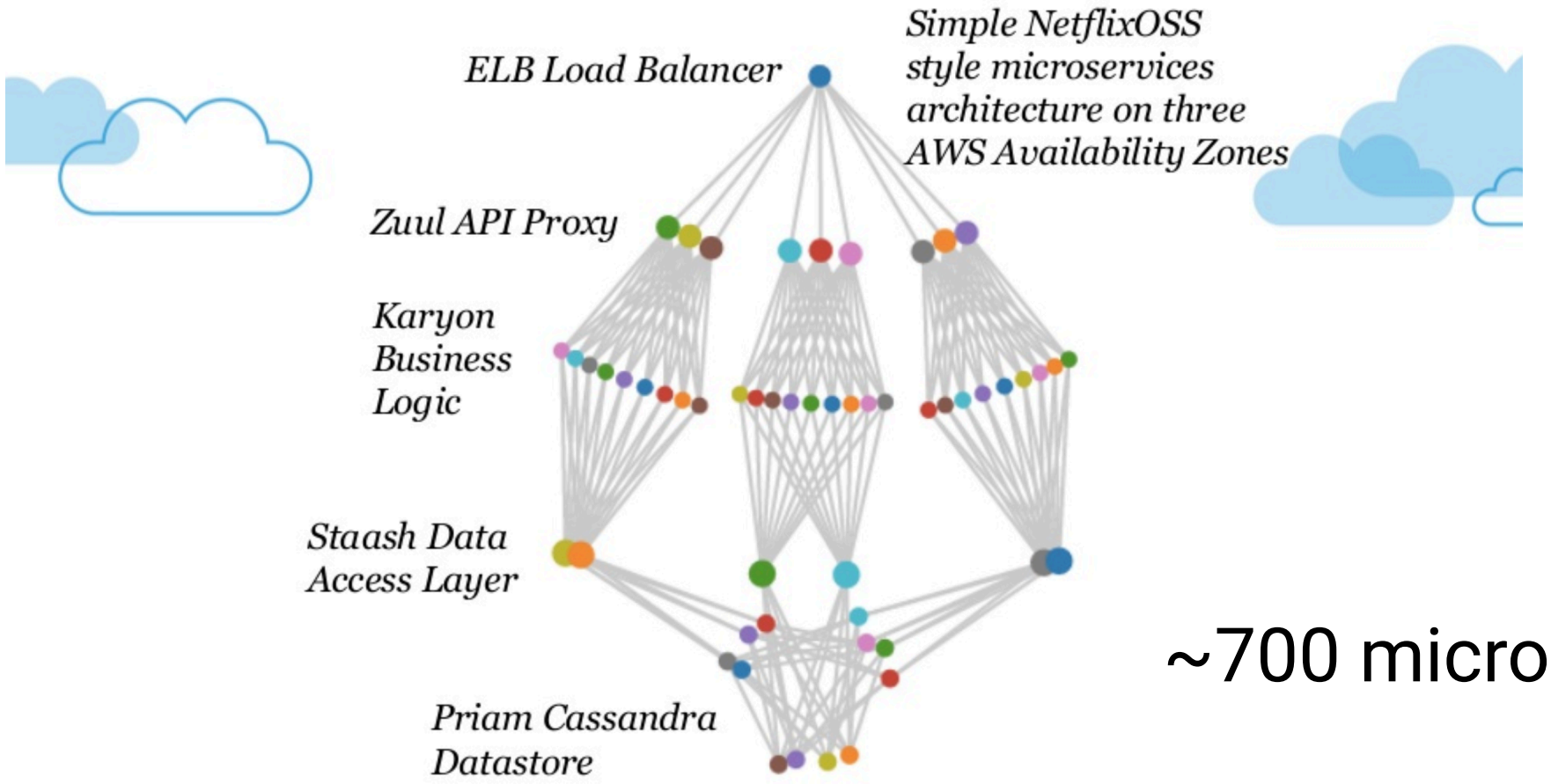
Computer systems cannot efficiently support *Microsecond-scale service time*



Computer systems cannot efficiently support *Microsecond-scale service time*



Microservice Example



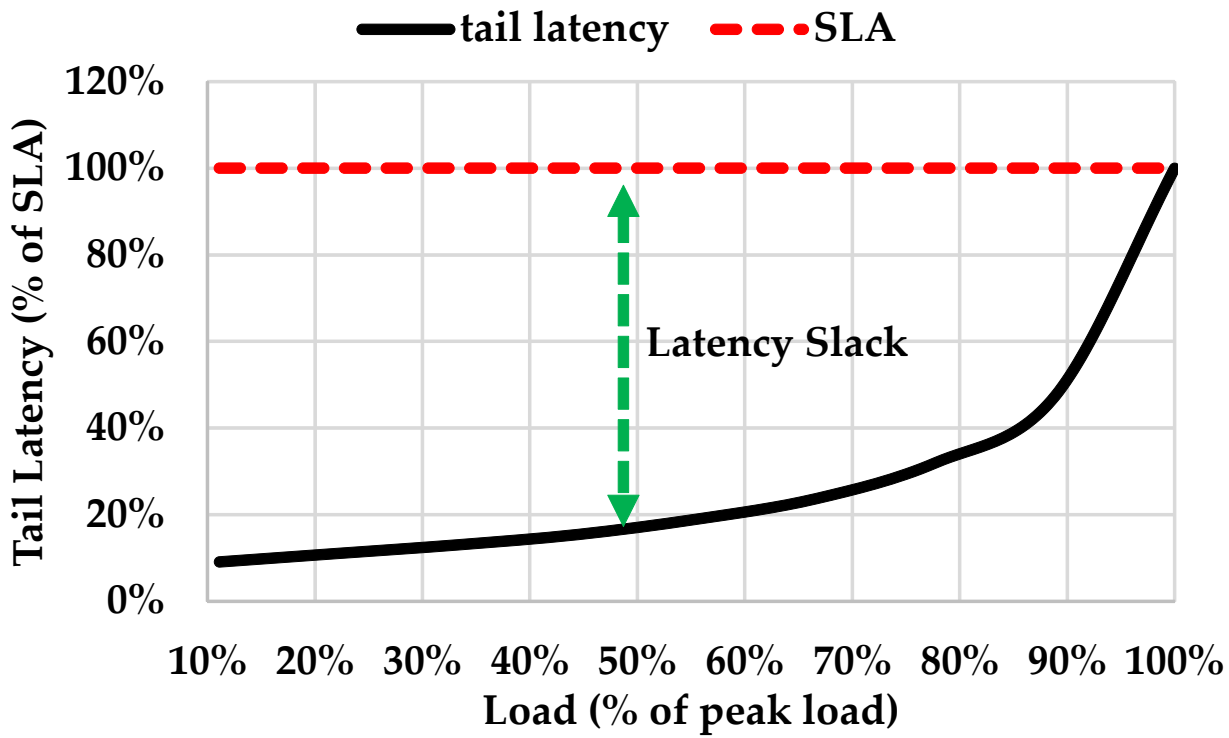
~700 microservices

Source: Adrian Cockcroft, "Monitoring Microservices and Containers: A Challenge"

Implications of *Killer Microsecond service time* on Dynamic Power Management?

Opportunity for DPM – Latency Slack

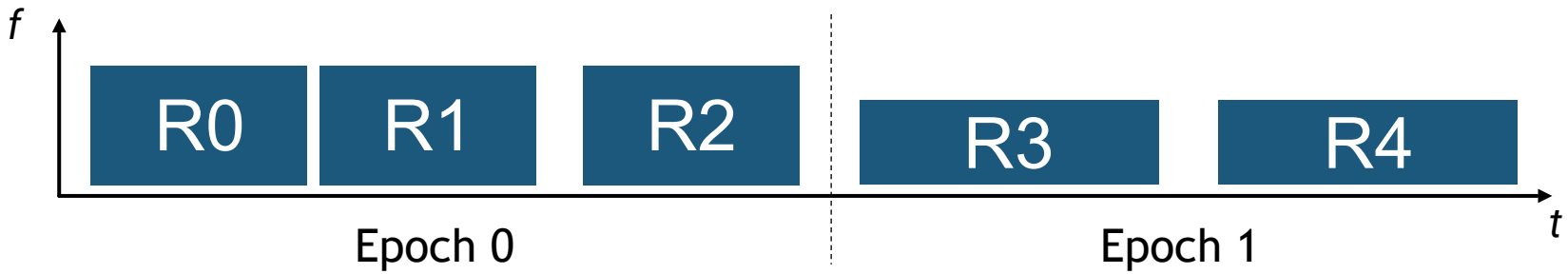
- Slow down request processing (DVFS)
- Delay request processing (Sleep)



Dynamic Power Management Overview

- DVFS (Rubik [MICRO'15], Pegasus [ISCA'14])
  - Rubik adjusts *f* per request

- DVFS + Sleep
  - SleepScale [ISCA'14] finds optimal frequency & C-state depth for 60s epochs

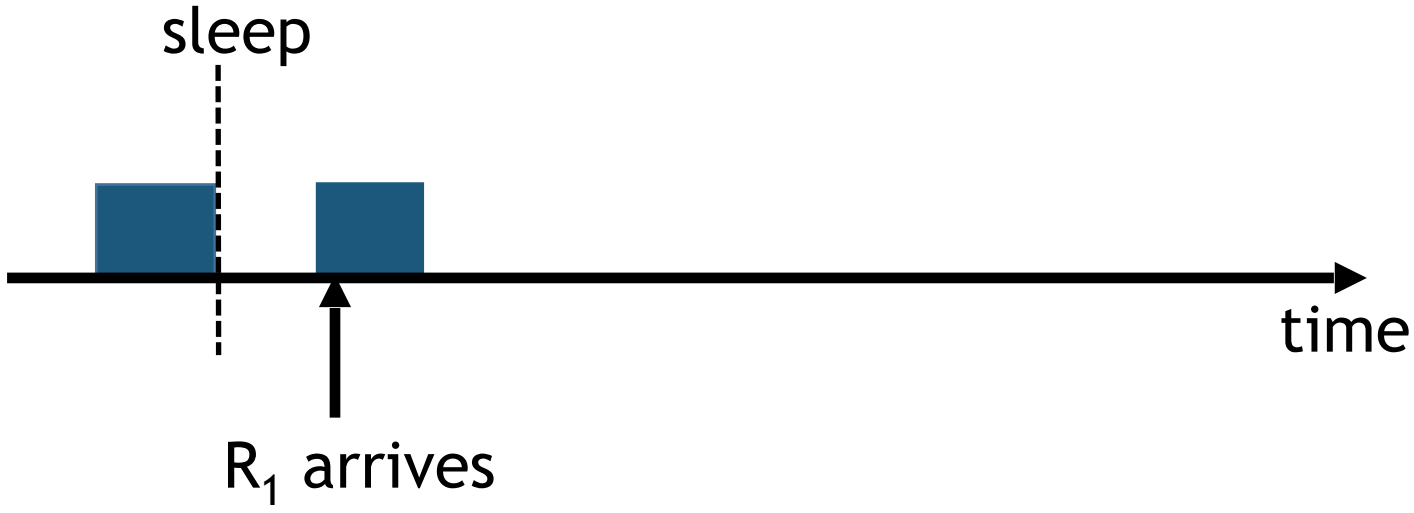


Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])

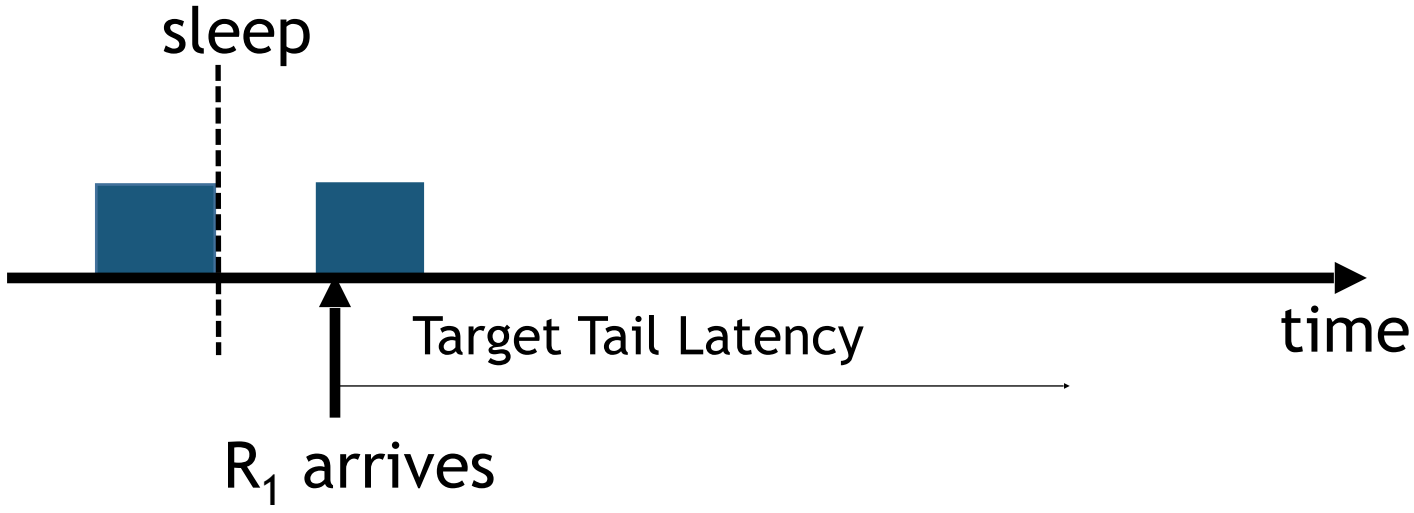
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



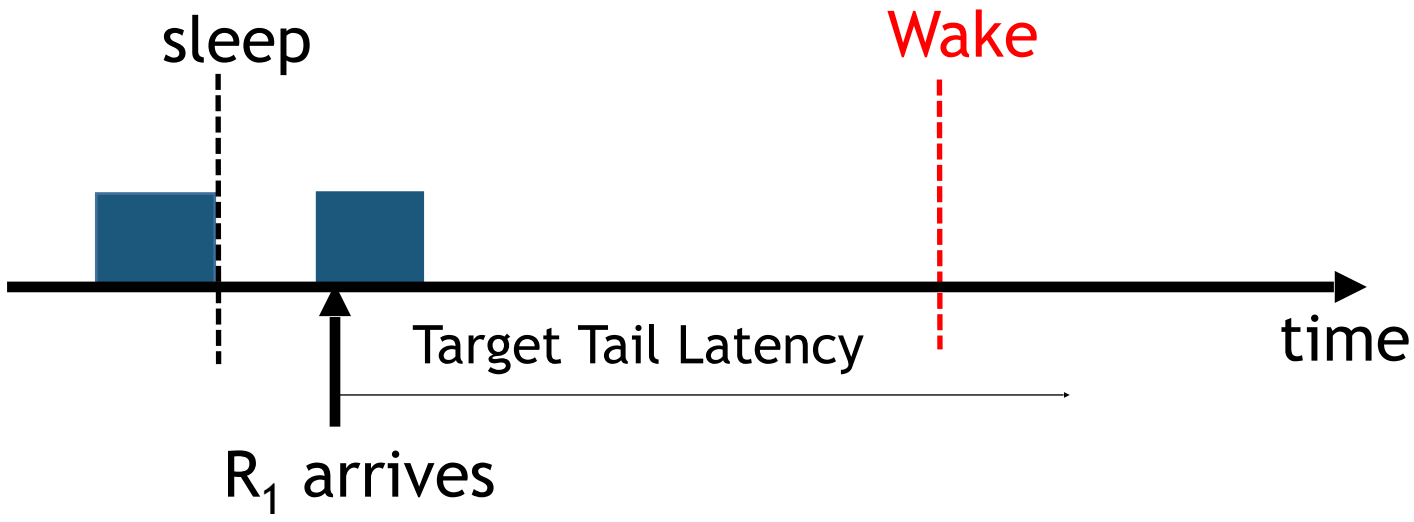
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



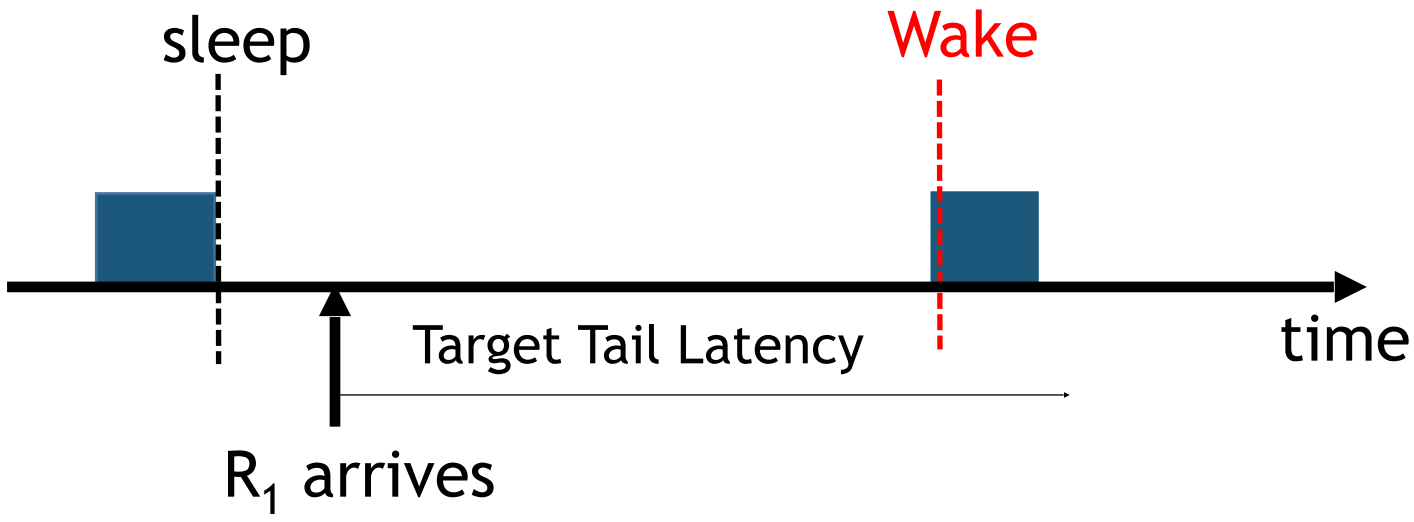
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



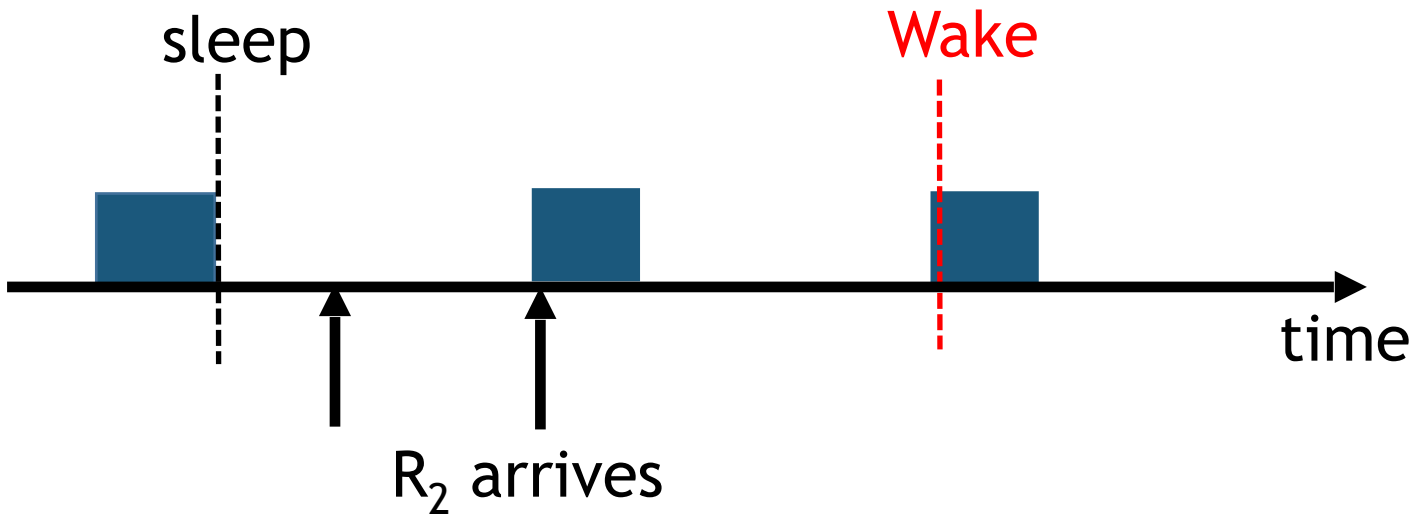
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



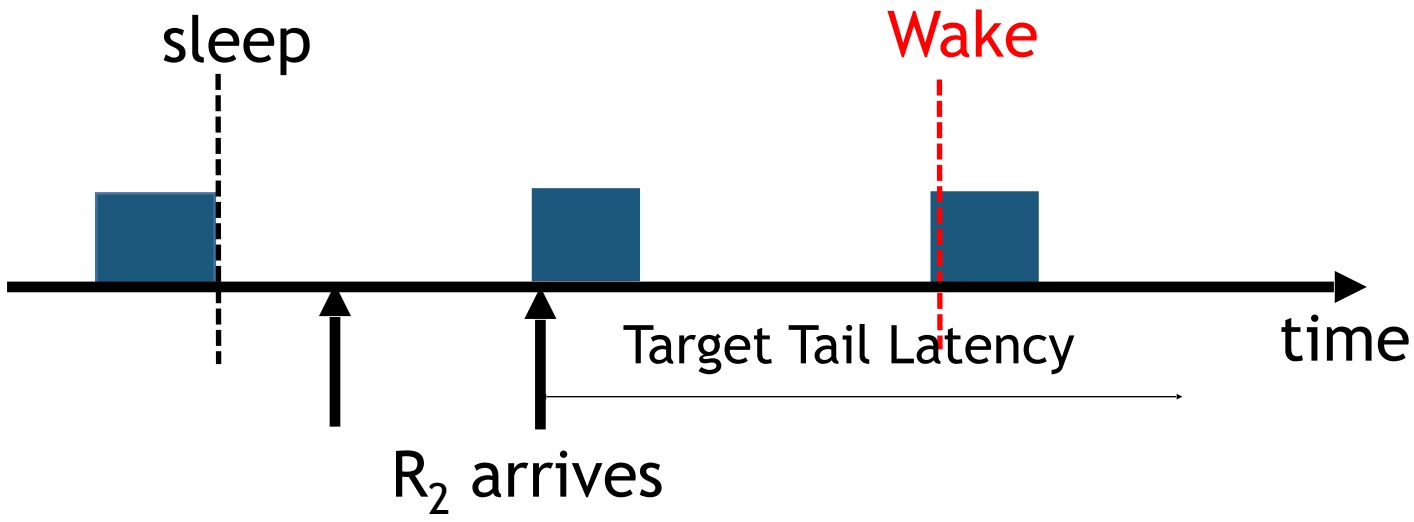
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



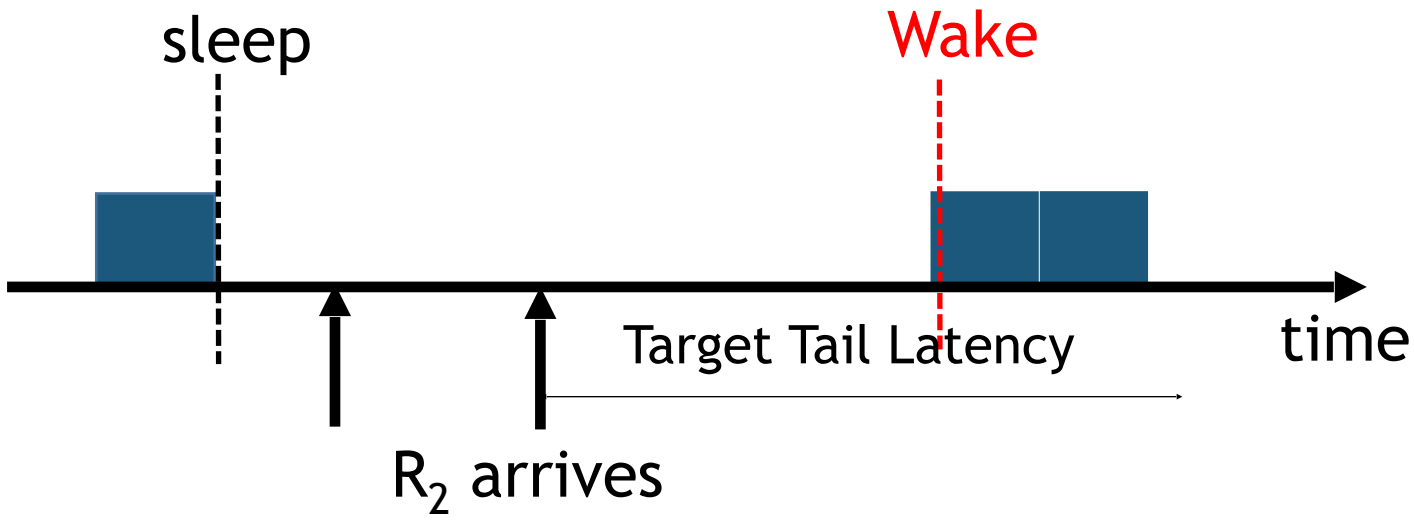
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



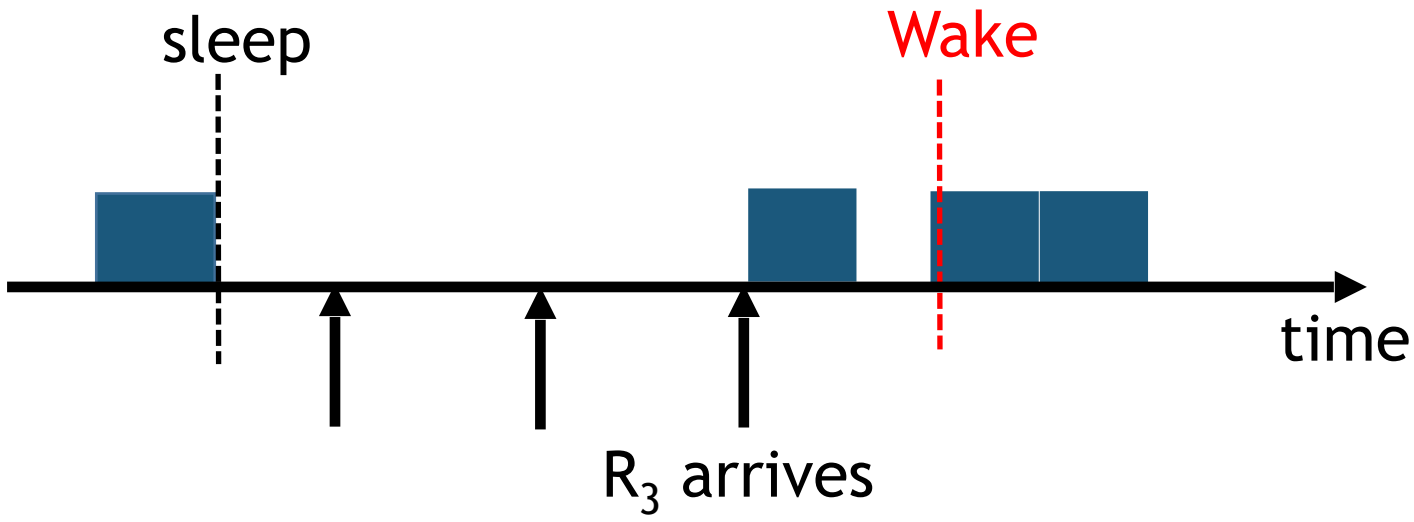
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



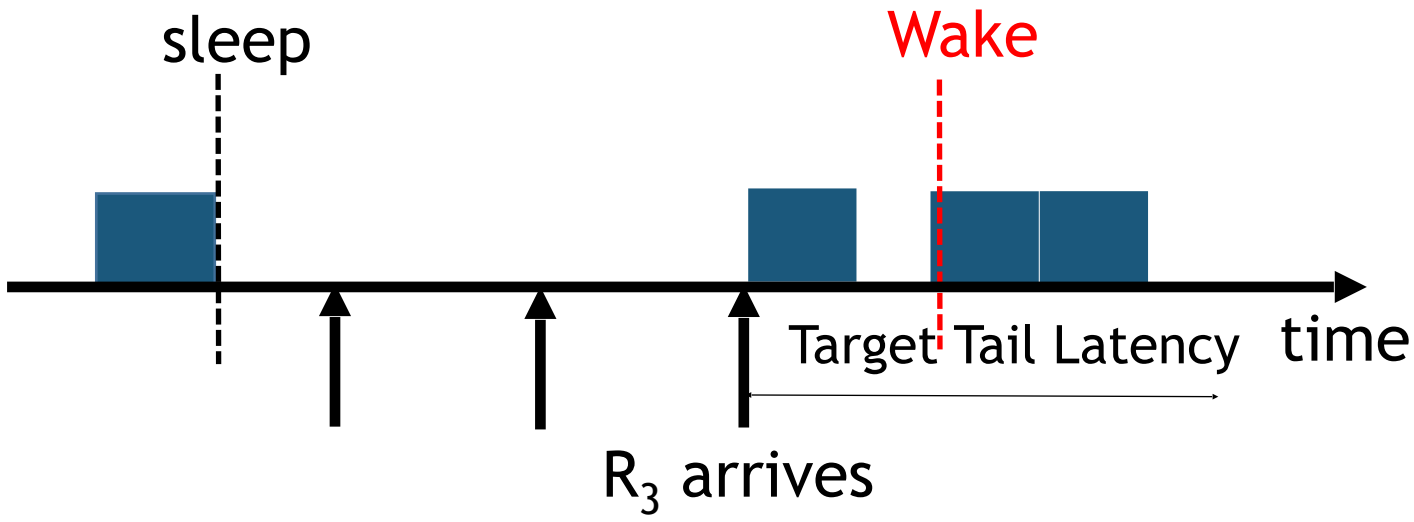
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



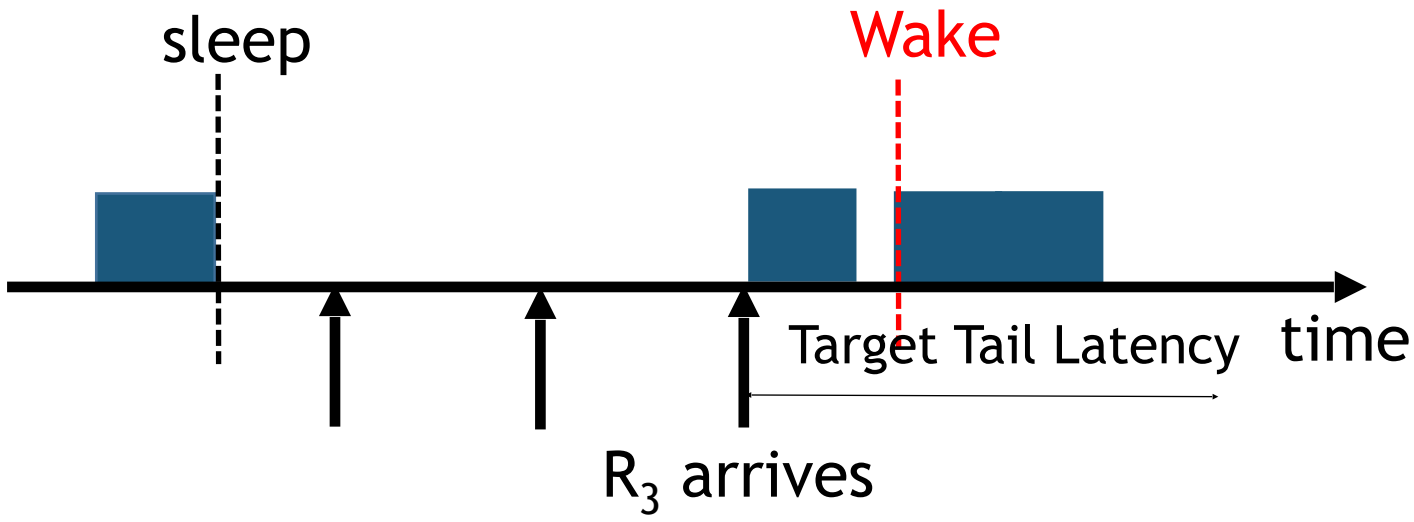
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



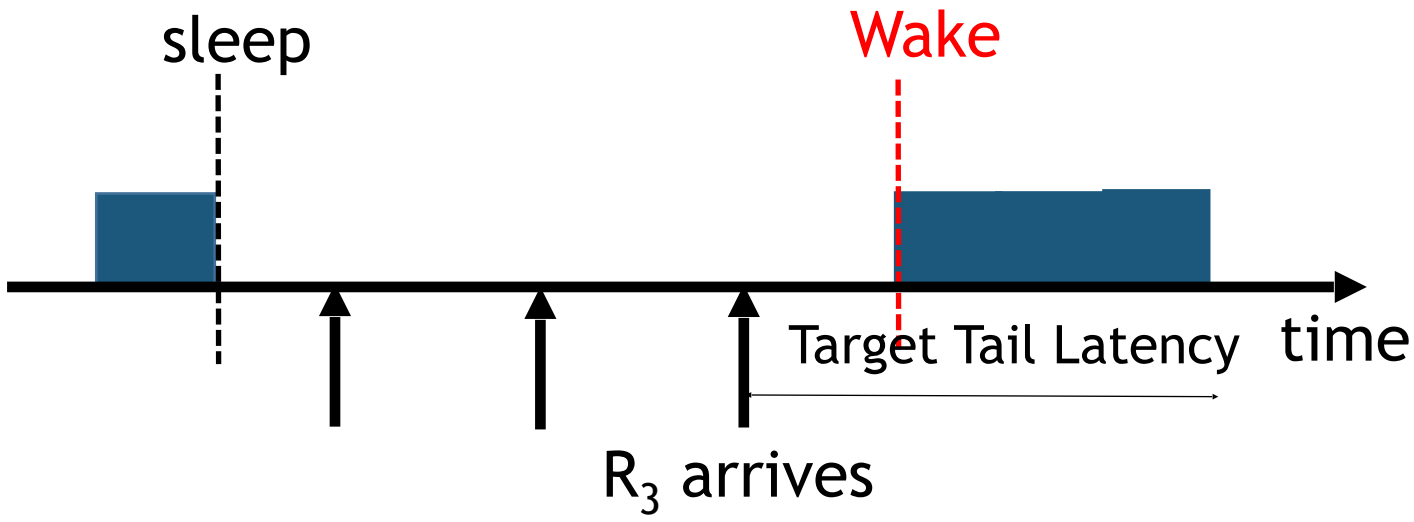
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



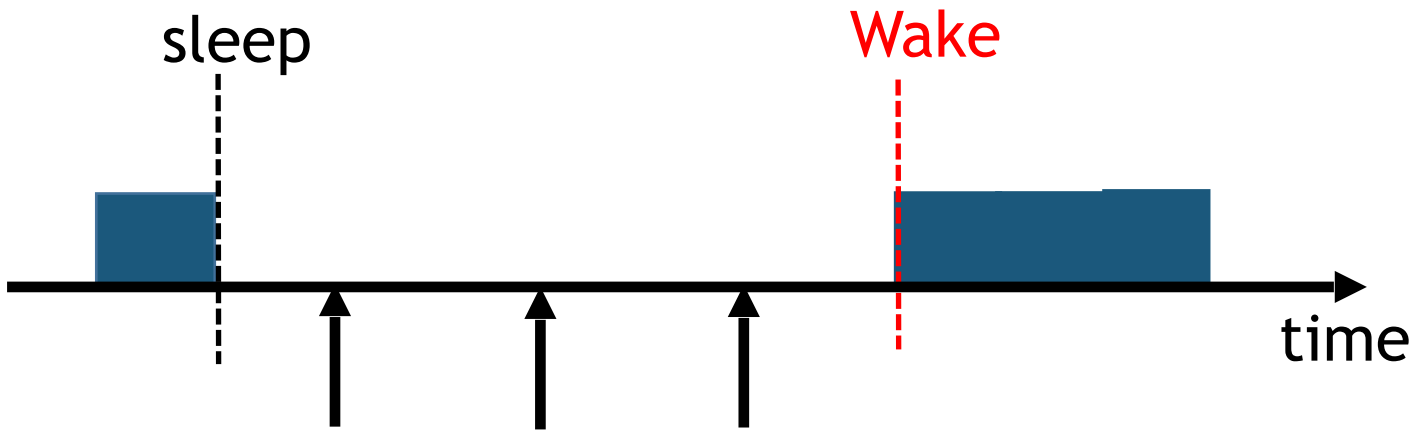
Dynamic Power Management Overview

- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])

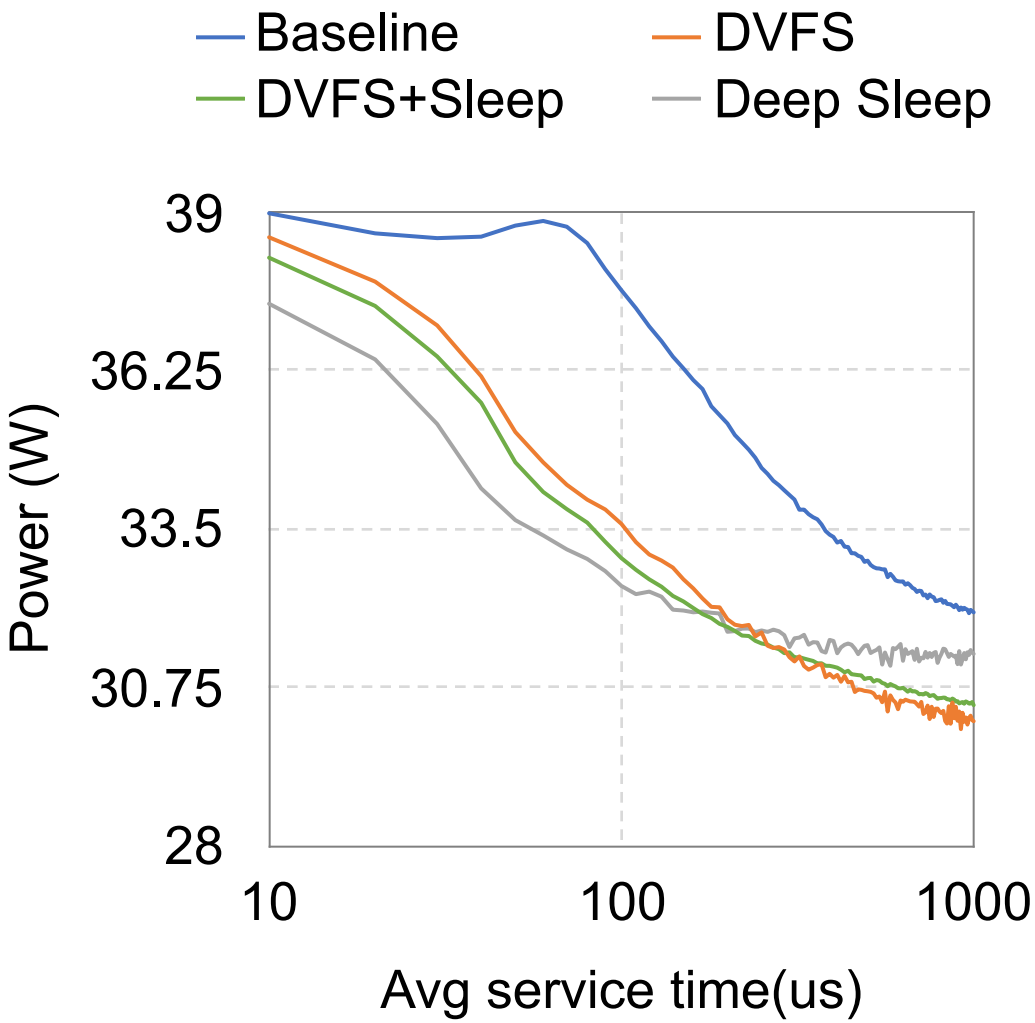


Dynamic Power Management Overview

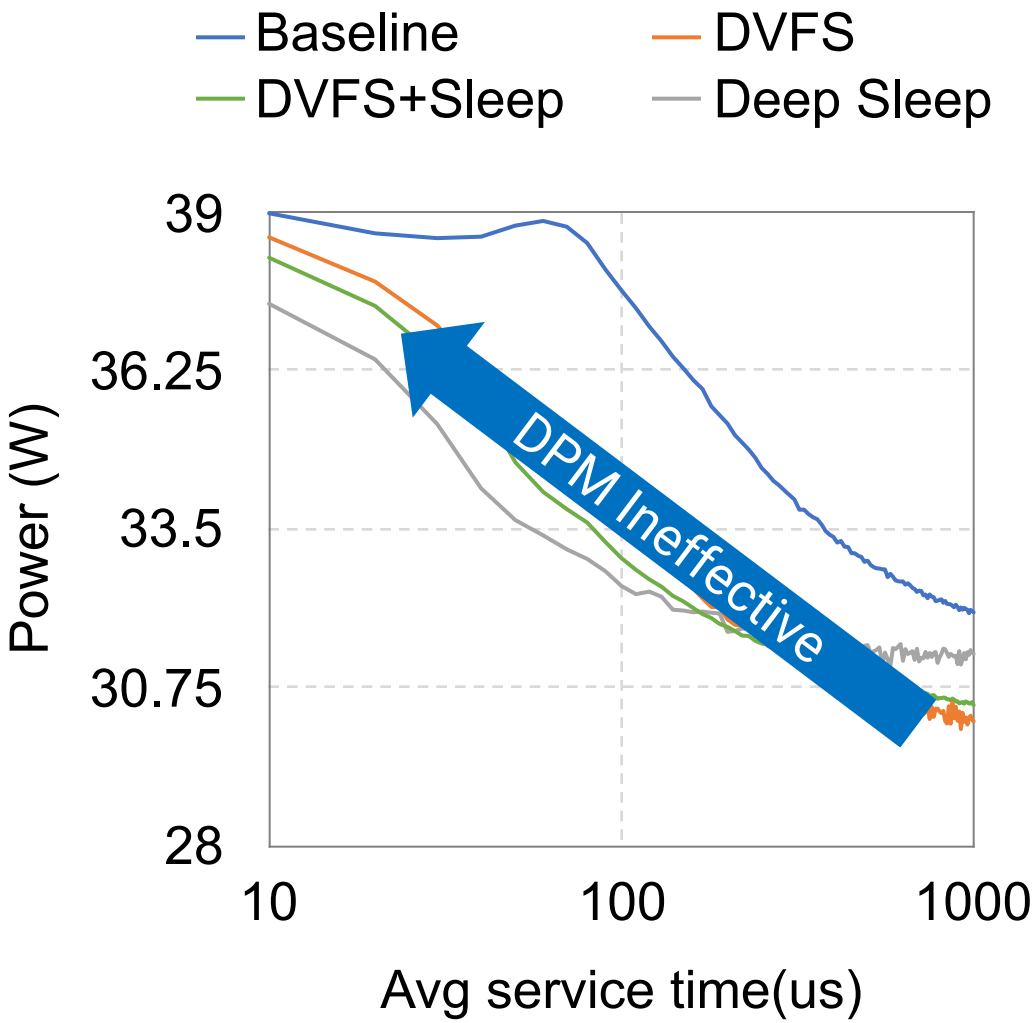
- Sleep states (PowerNap[ASPLOS'09], Dreamweaver[ASPLOS'12], DynSleep[ISLPED'16], CARB[CAL'17])



DPM ineffective w/ microsecond service time

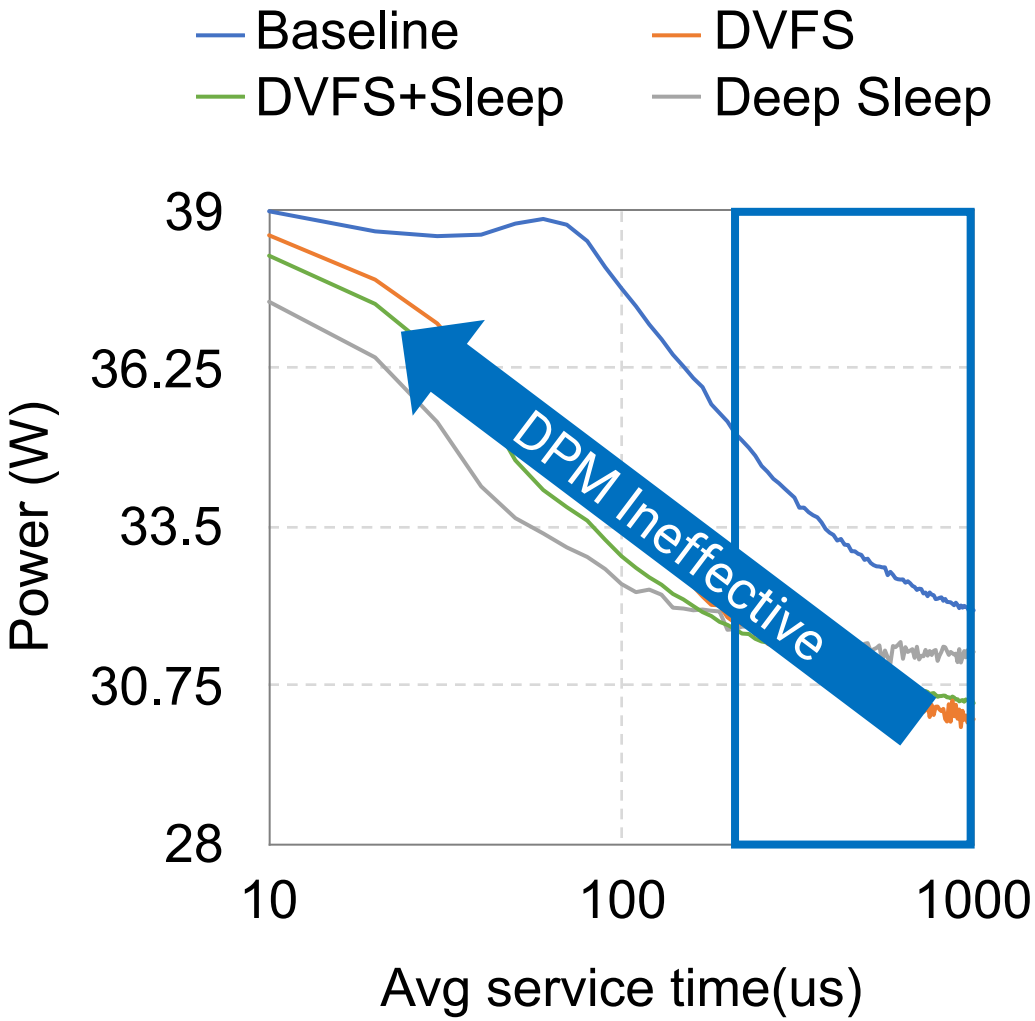


DPM ineffective w/ microsecond service time



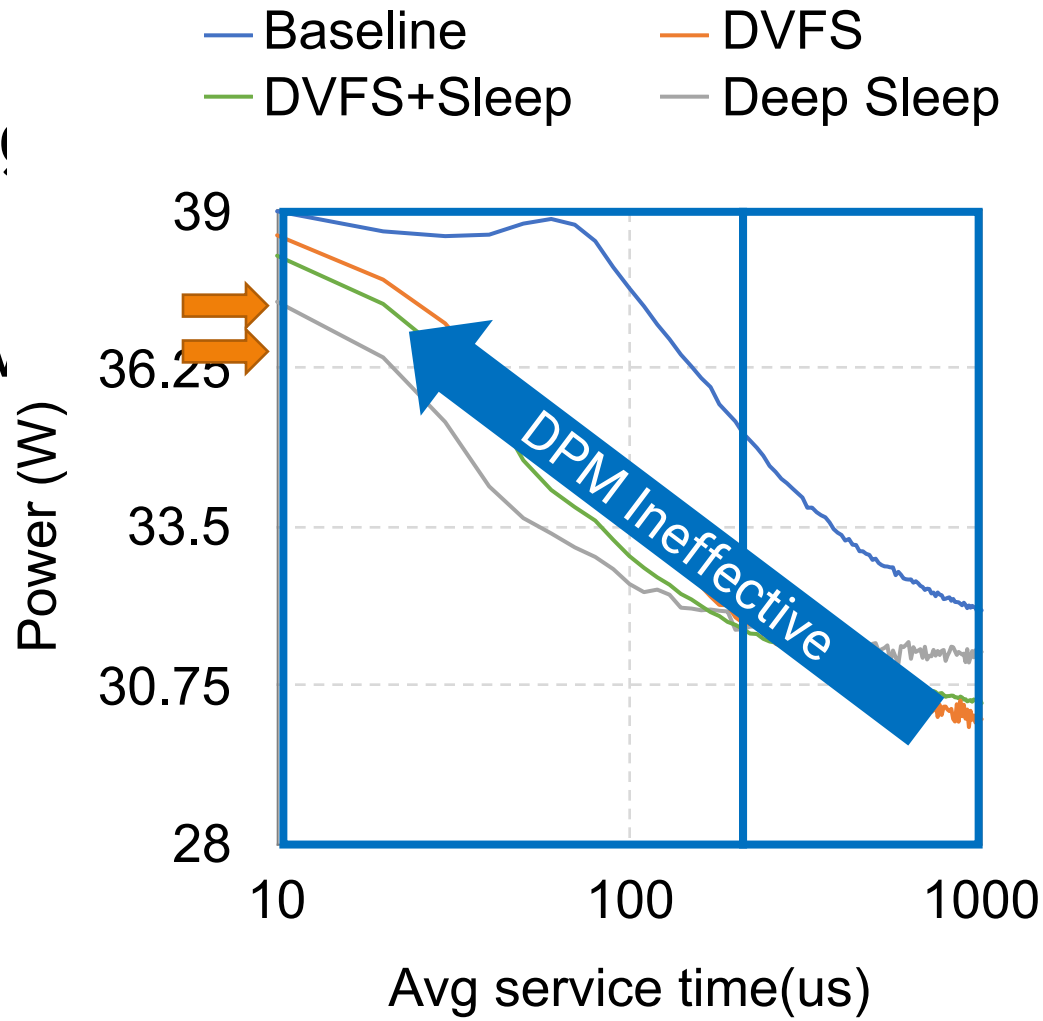
DPM ineffective w/ microsecond service time

- >250µs: DVFS effective at slowing down request processin



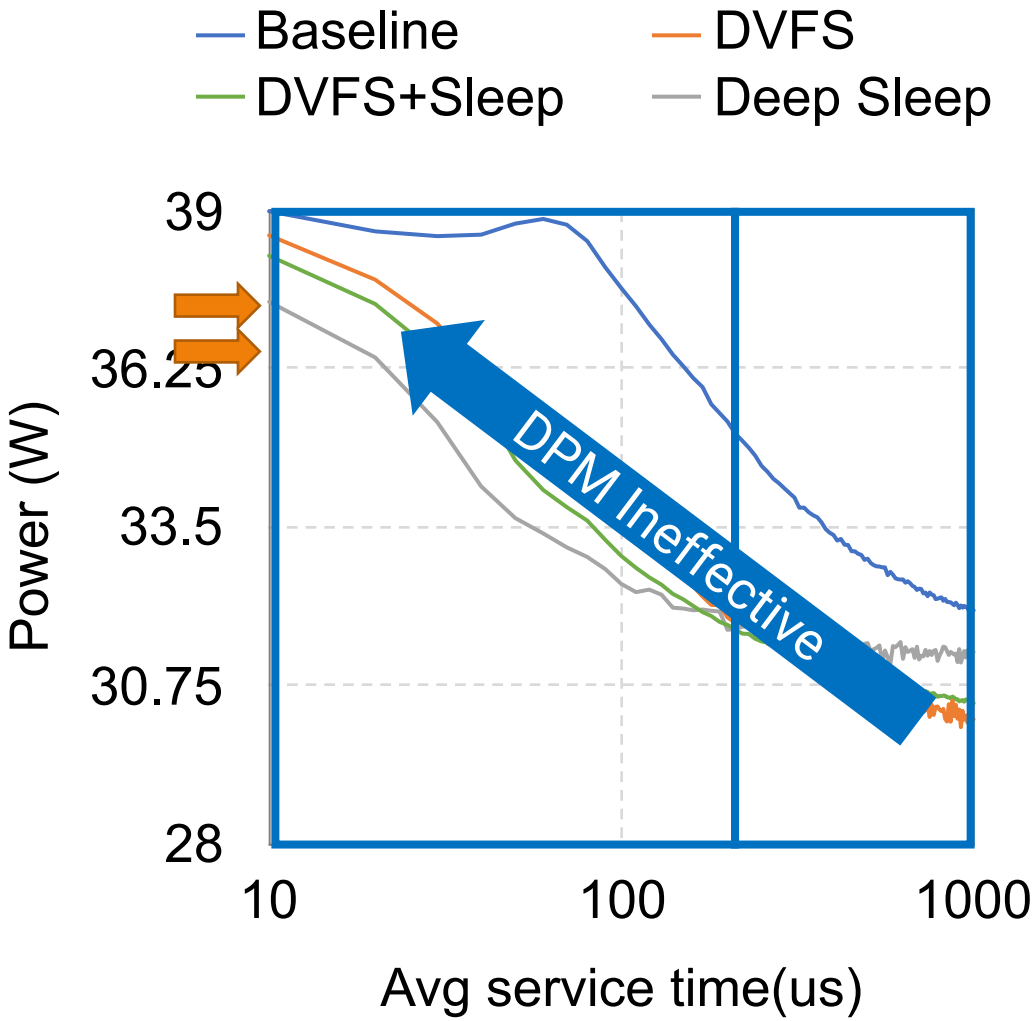
DPM ineffective w/ microsecond service time

- >250µs: DVFS effective at slowing down request processin
- <250µs: DPM becomes ineffectiv

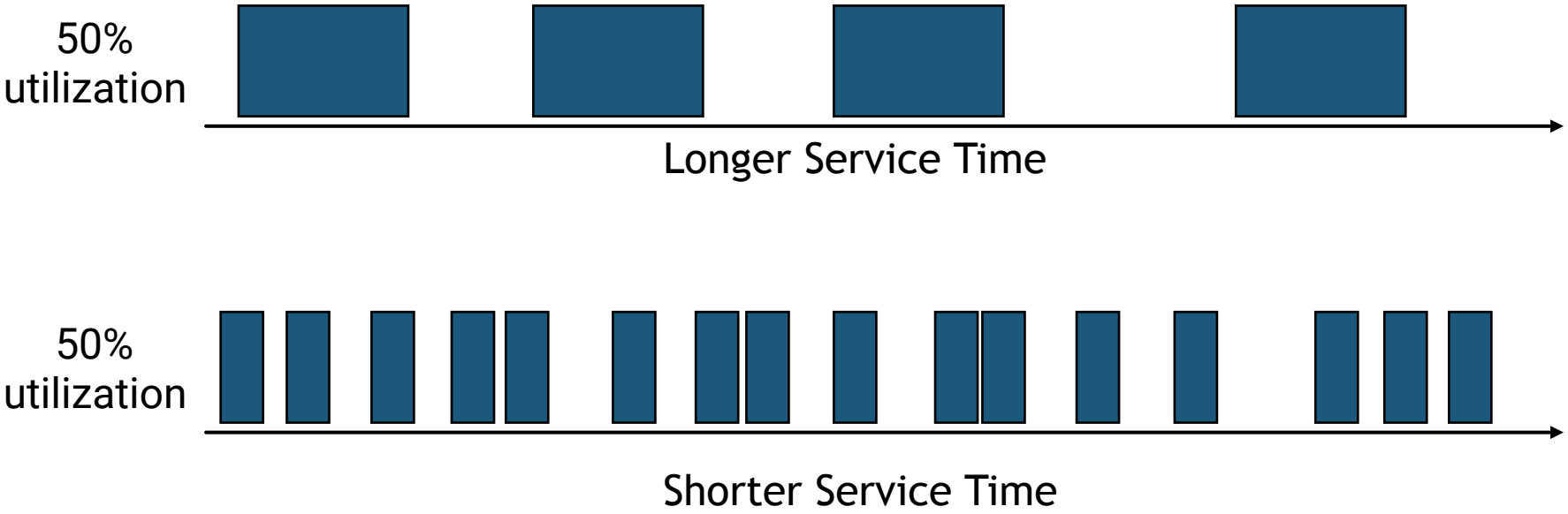


DPM ineffective w/ microsecond service time

- >250µs: DVFS effective at slowing down request processin
- <250µs: DPM becomes ineffecti
- Surprisingly, sleep-based policies outperform DVFS-based policies

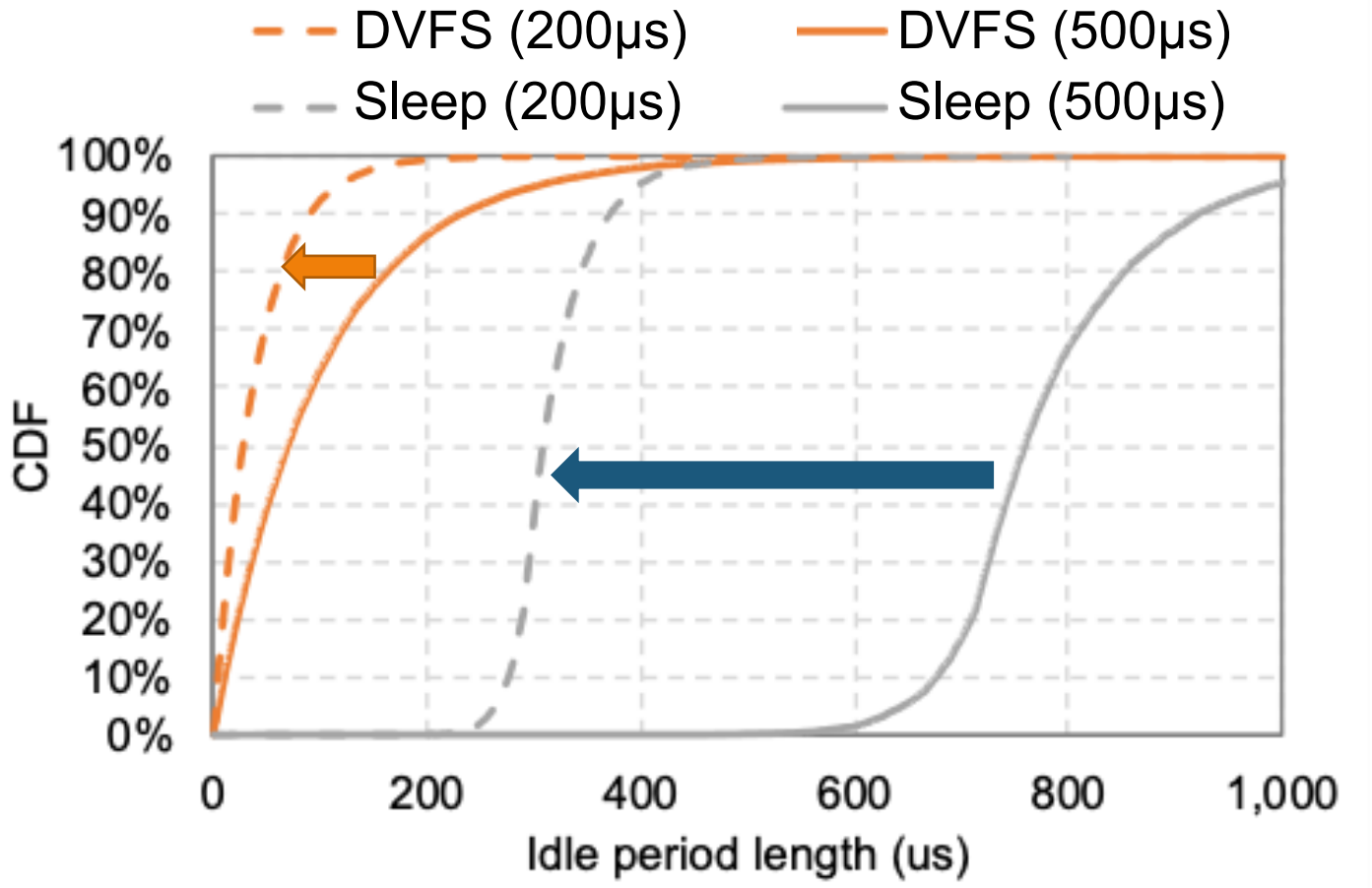


Fragmented idle periods → Lost Opportunities



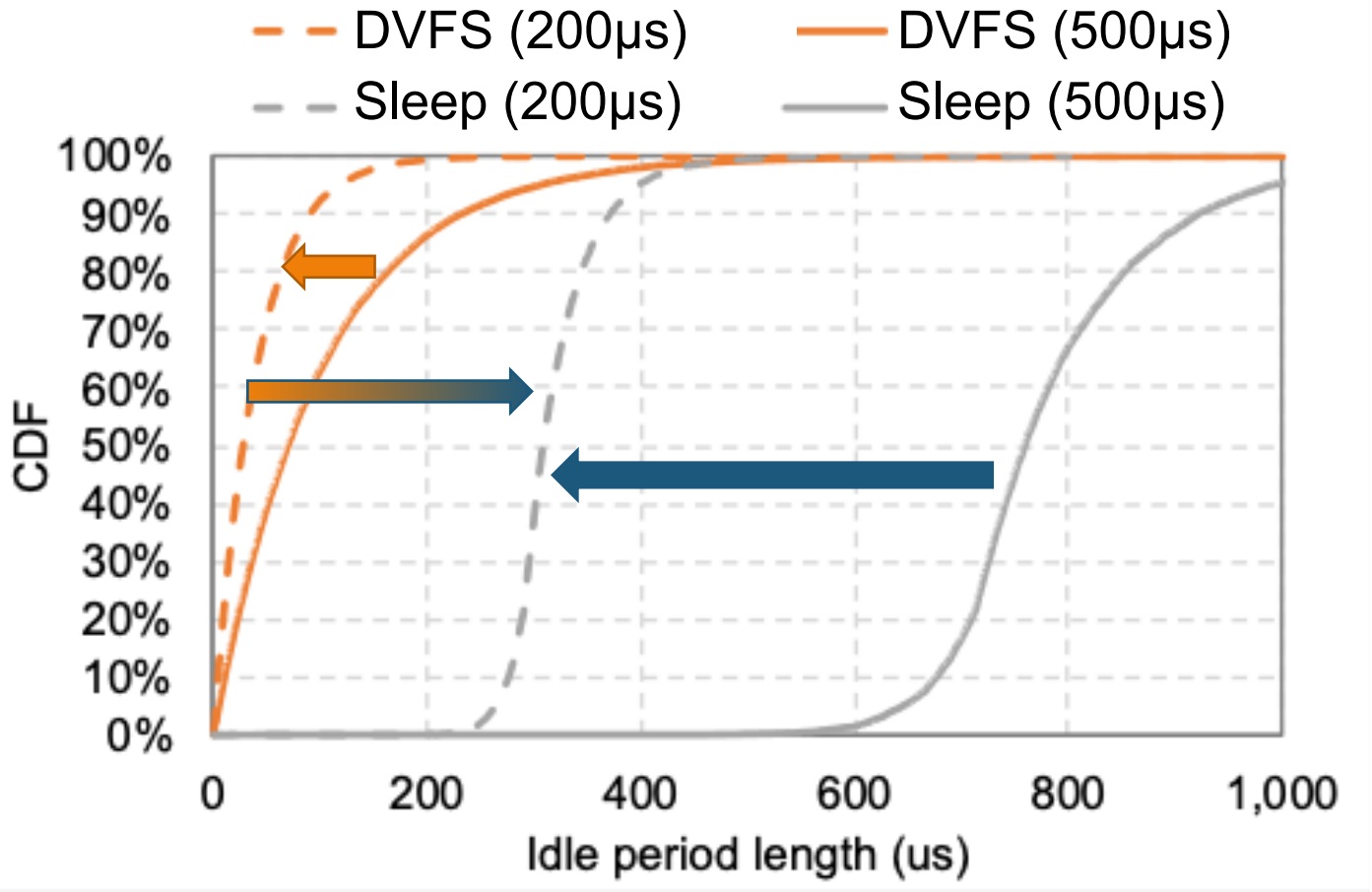
Fragmented idle periods → Lost Opportunities

- Short service times fragment idle periods



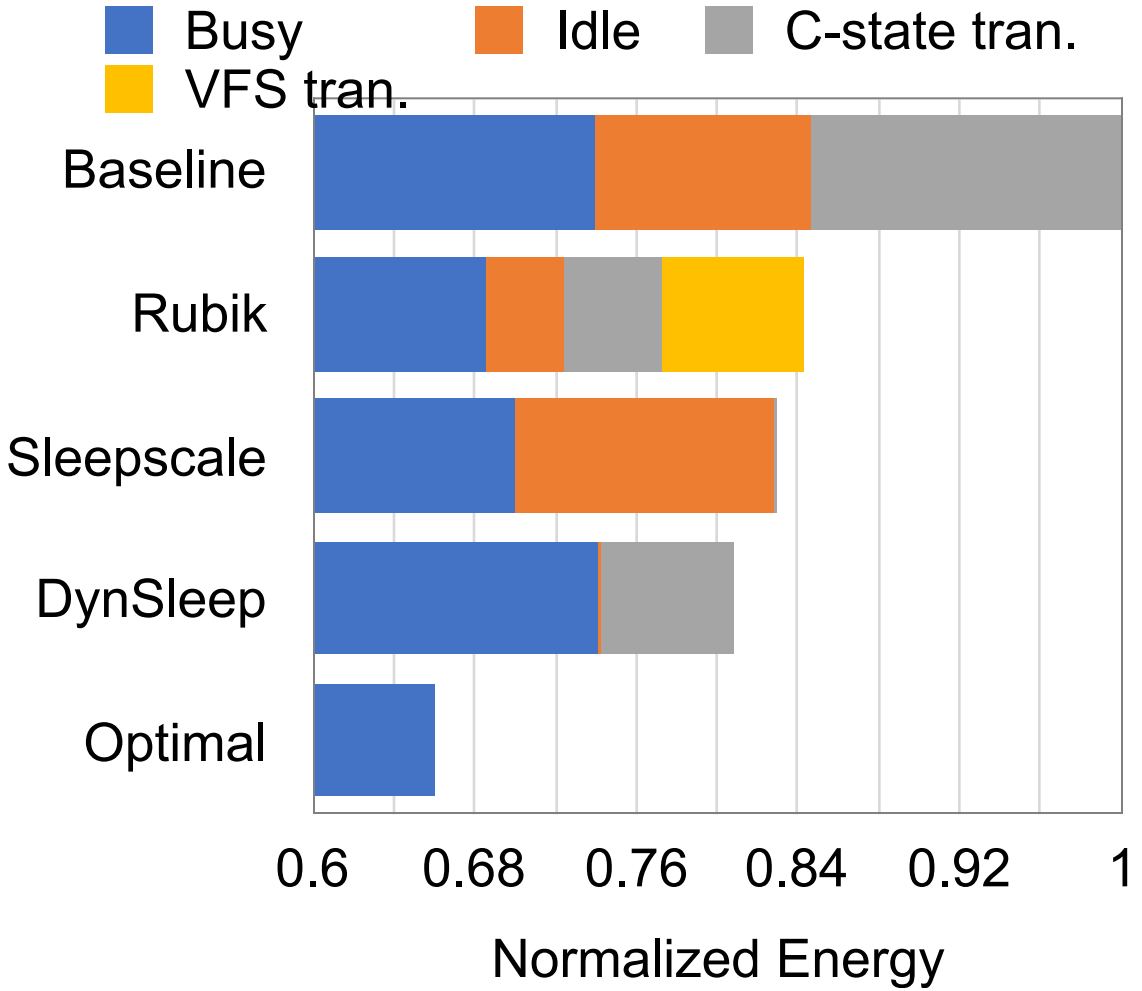
Fragmented idle periods → Lost Opportunities

- Short service times fragment idle periods
- Sleep states / request delaying can consolidate idle periods

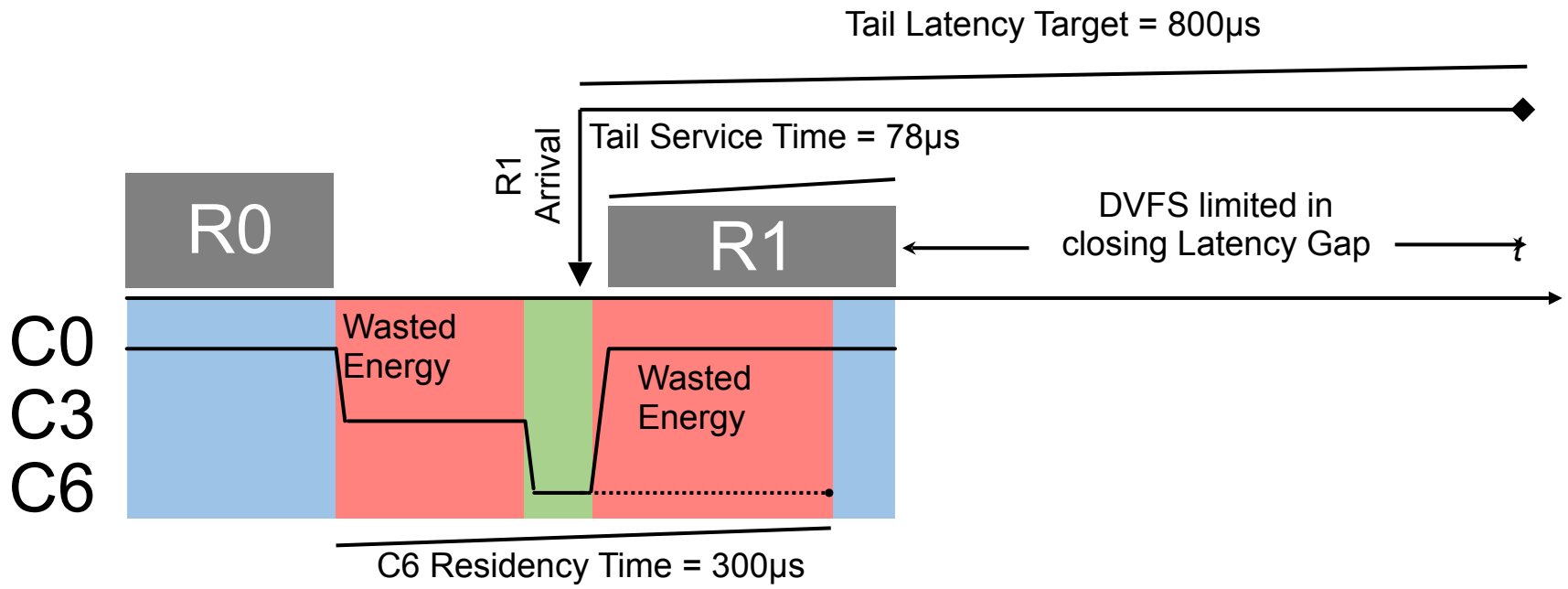


Significant transition overheads and idle power

Idleness and transition overhead still account for up to ~25% of energy

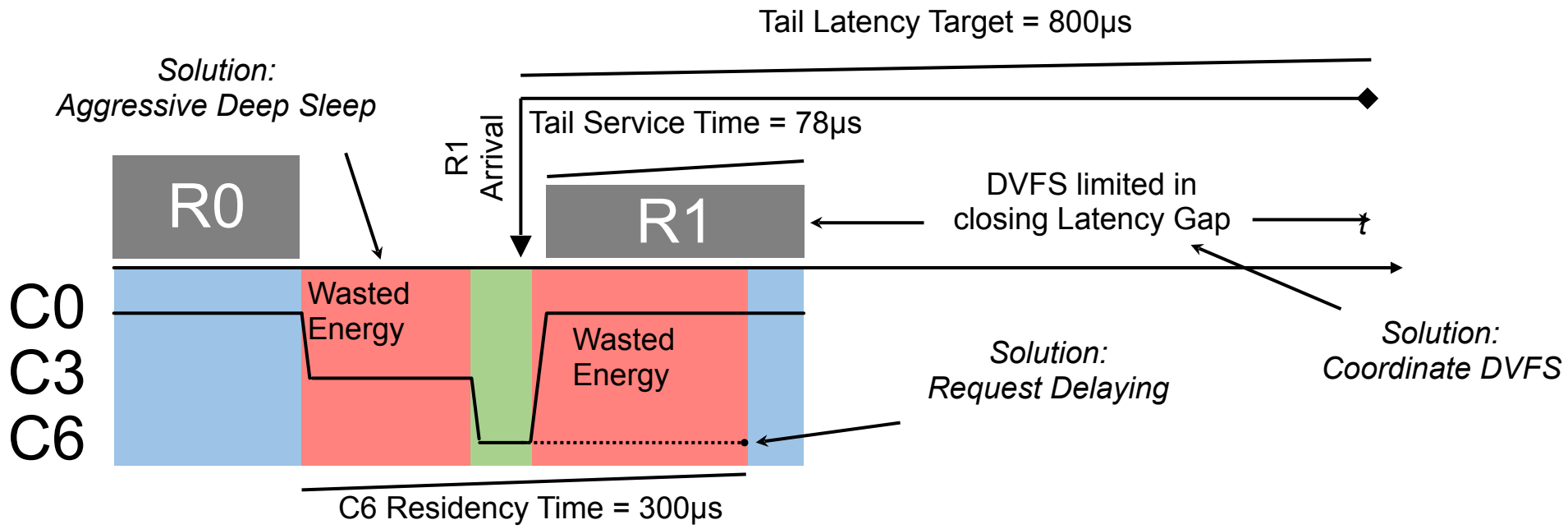


DPM inefficiencies



* SPECjbb timing

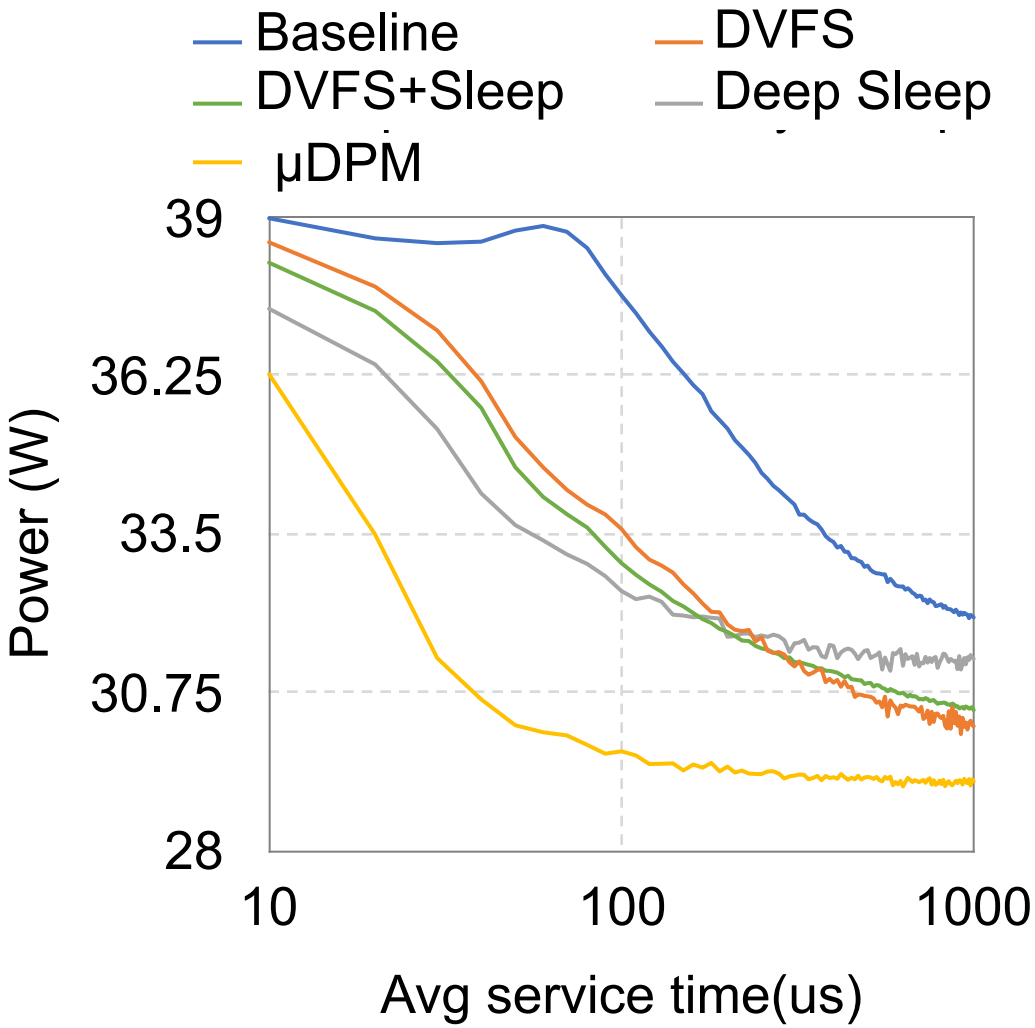
DPM inefficiencies



* SPECjbb timing

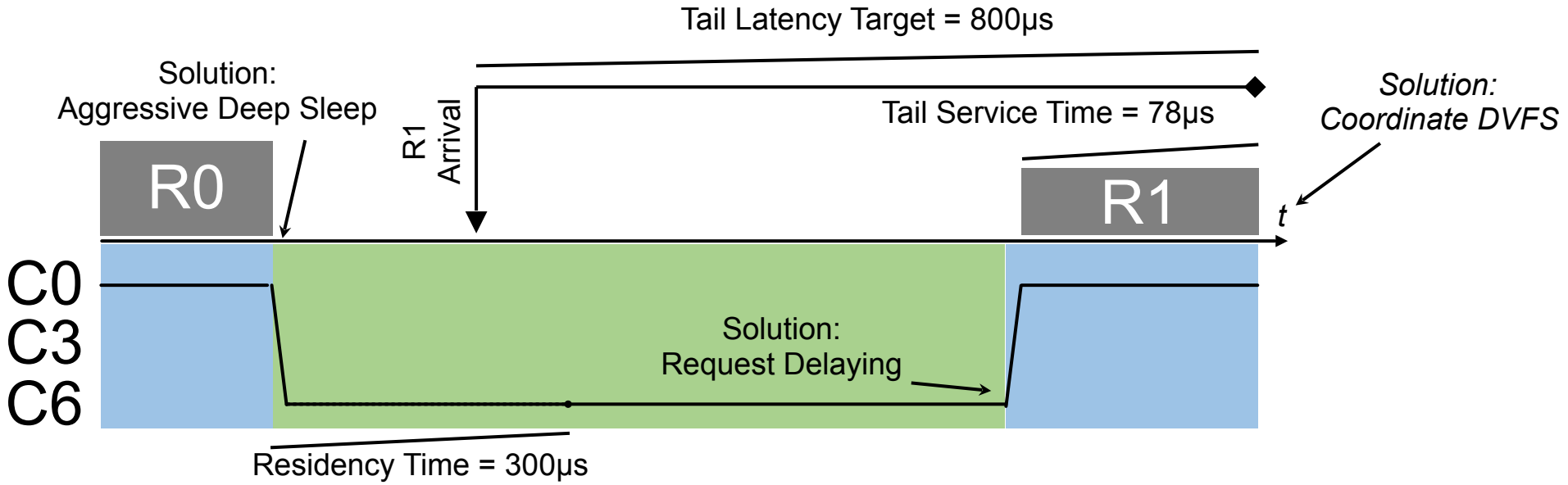
Key Insight

Careful coordination of DVFS, Sleep state, and request delaying is the key to effective DPM with microsecond service times

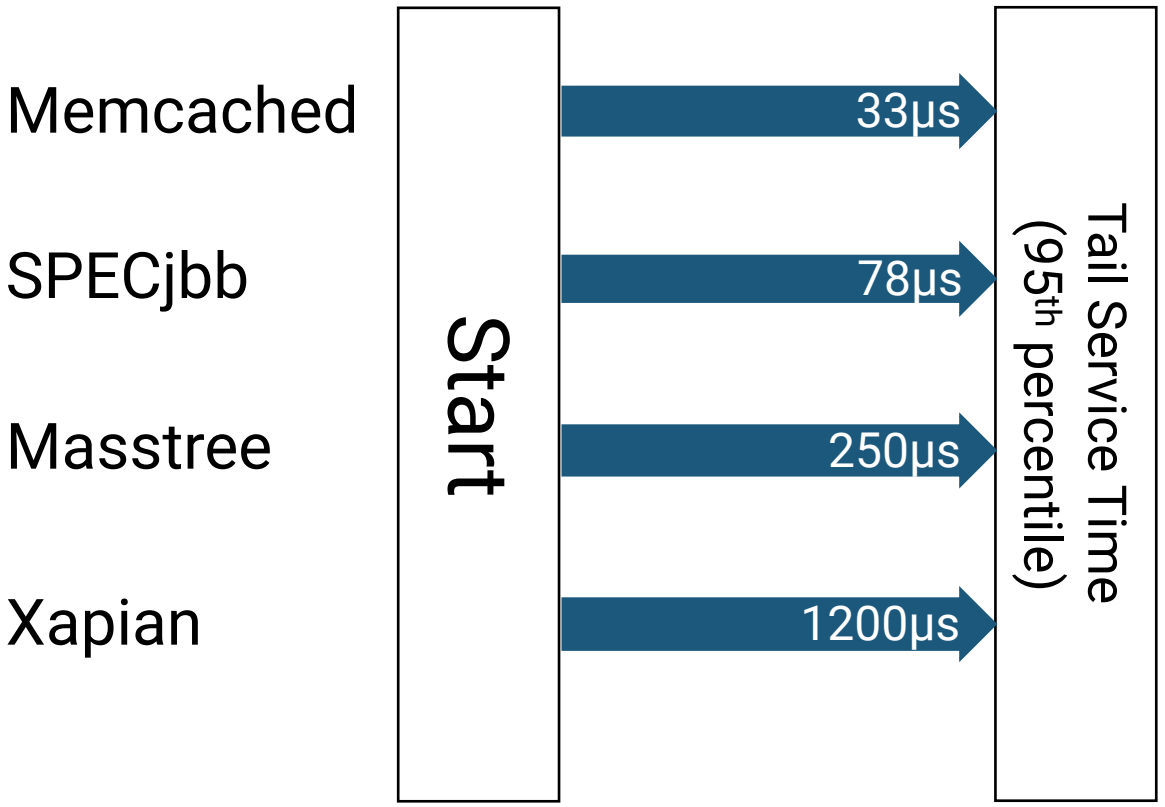


µDPM

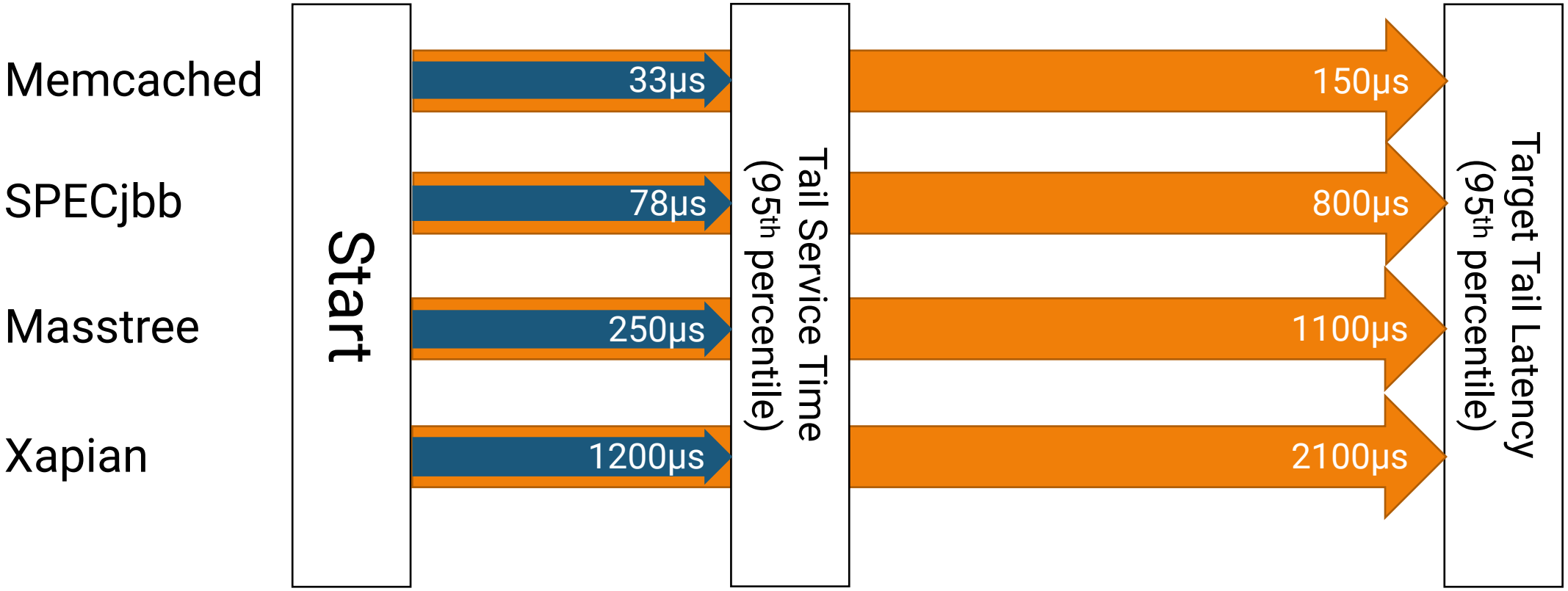
- Aggressively Deep Sleep
- Delay and slow down request processing to finish just-in-time, even under microsecond request service times
- Carefully coordinating DVFS, Sleep, and request delaying



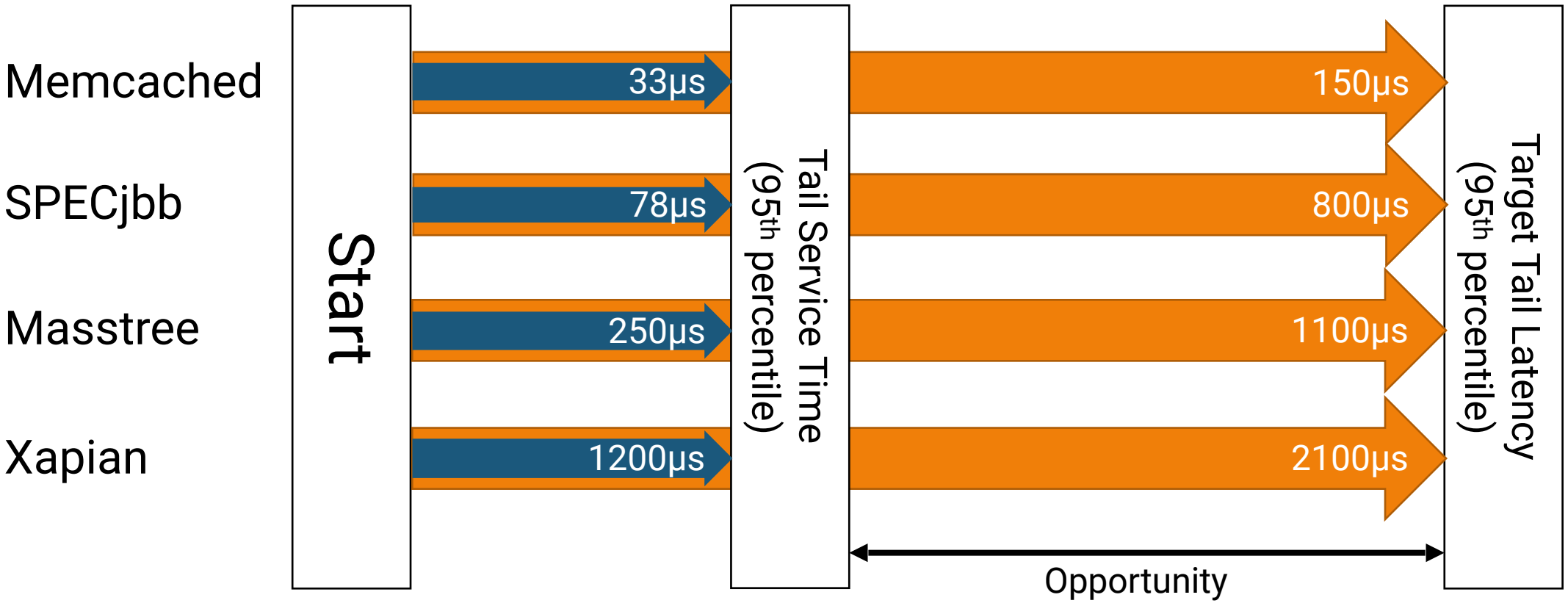
Can latency-critical workloads utilize deep sleep states?



Can latency-critical workloads utilize deep sleep states?

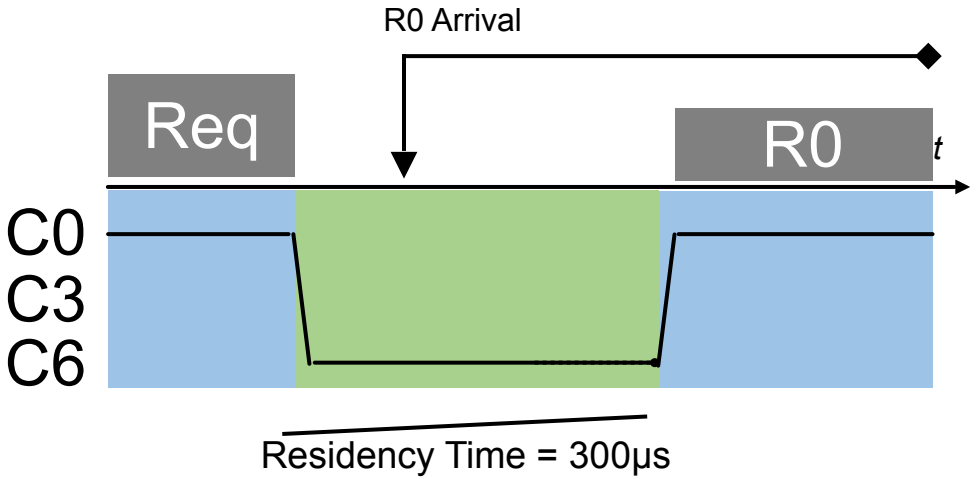


Can latency-critical workloads utilize deep sleep states?



Aggressive deep sleep and request delaying

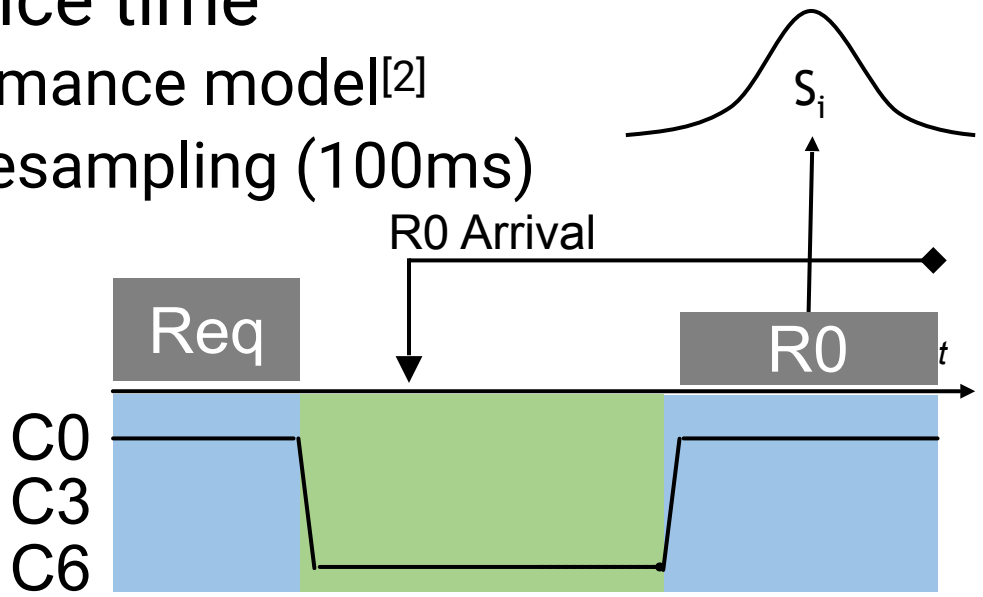
- Wakeup after residency time



- Wakeup before residency time if needed to meet tail latency

Estimating Service time and Latency

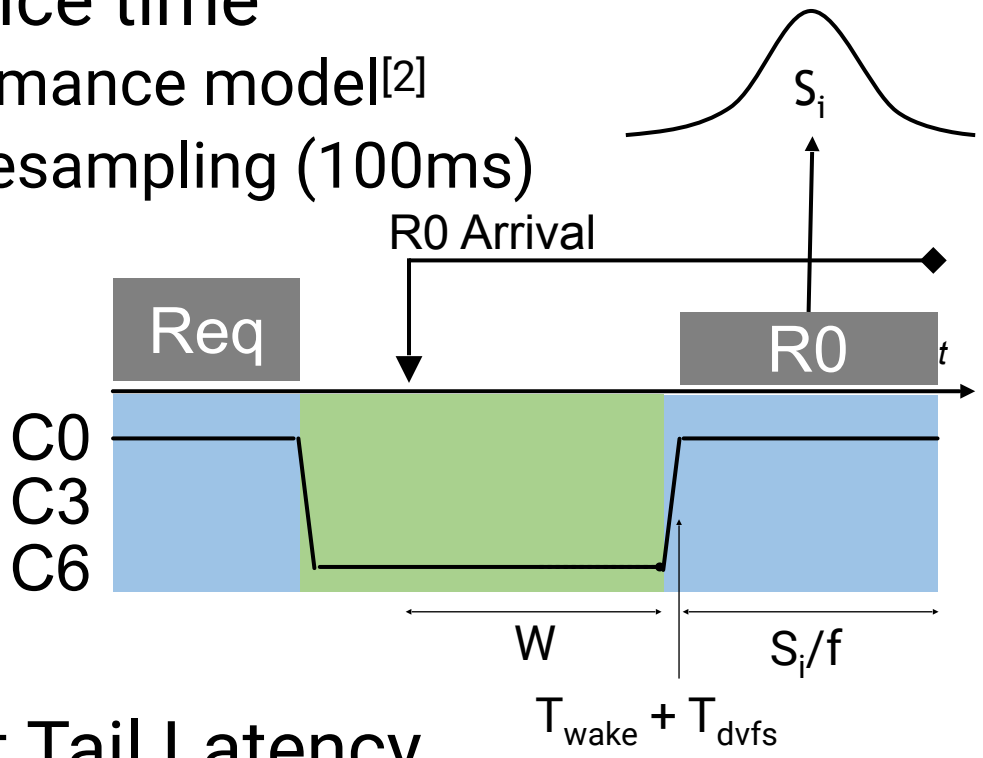
- Estimate tail service time
  - Statistical performance model[2]
  - Online periodic resampling (100ms)



[2] Kasture, Harshad, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. "Rubik: Fast analytical power management for latency-critical systems." MICRO 2015.

Estimating Service time and Latency

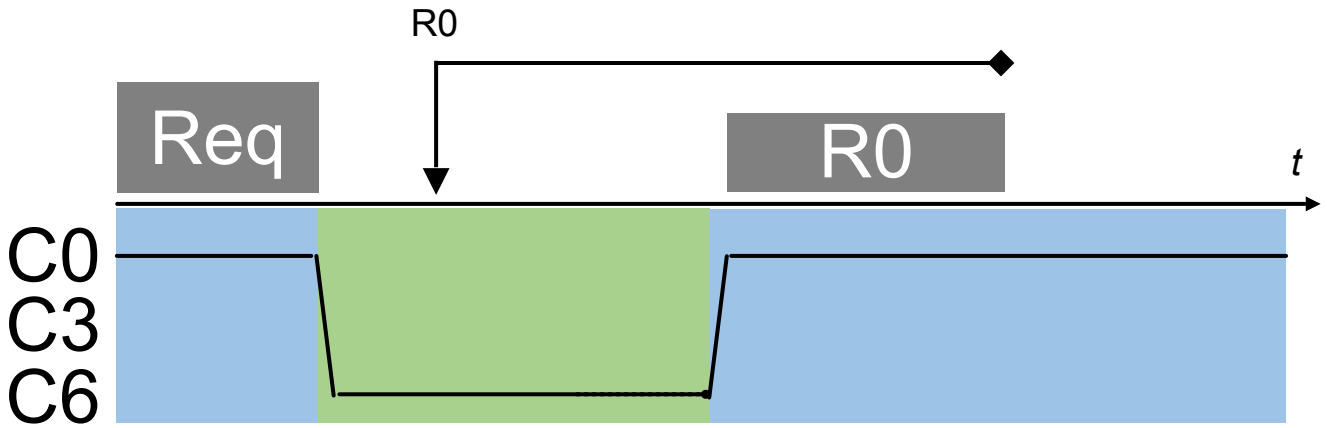
- Estimate tail service time
  - Statistical performance model[2]
  - Online periodic resampling (100ms)



- Estimate Request Tail Latency
  - $L = W + T_{wake} + T_{dvfs} + S_i / f$

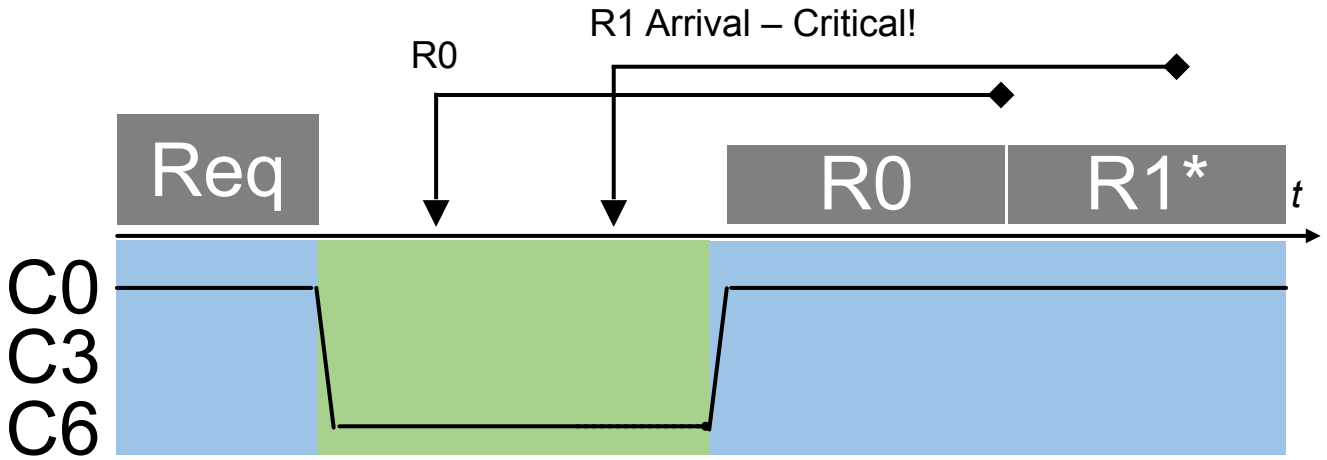
[2] Kasture, Harshad, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. "Rubik: Fast analytical power management for latency-critical systems." MICRO 2015.

Detecting critical request arrival



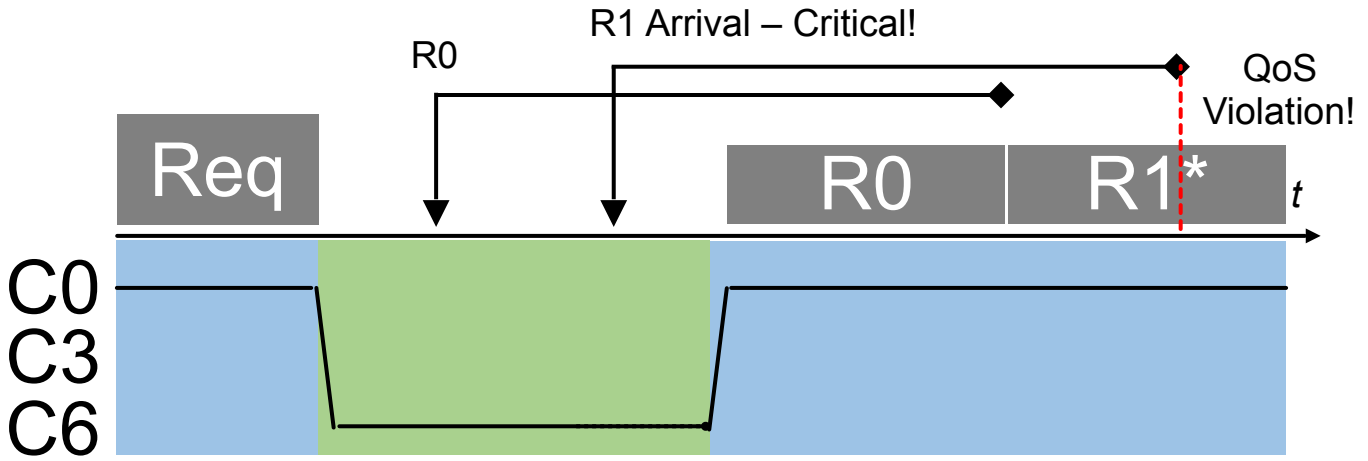
- Detecting critical request arrival
  - If inter-arrival time between 2 consecutive requests are shorter than the tail service time

Detecting critical request arrival



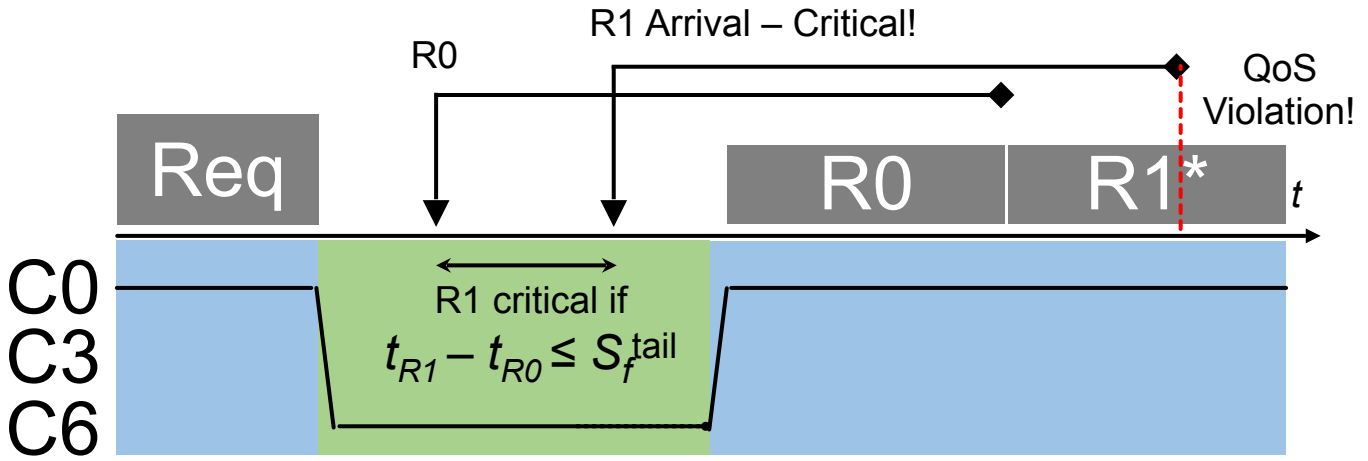
- Detecting critical request arrival
  - If inter-arrival time between 2 consecutive requests are shorter than the tail service time

Detecting critical request arrival



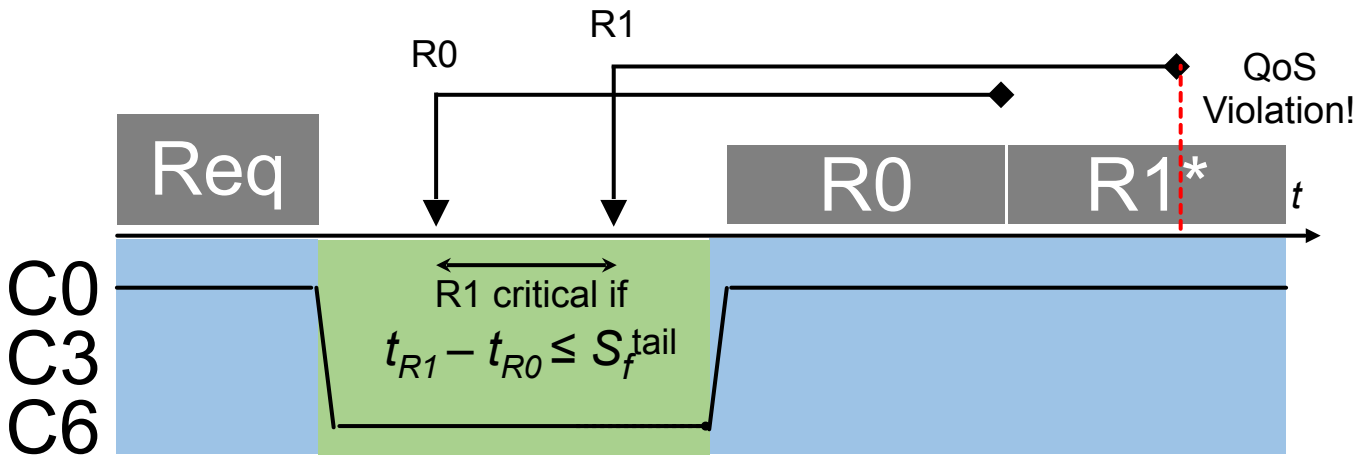
- Detecting critical request arrival
  - If inter-arrival time between 2 consecutive requests are shorter than the tail service time

Detecting critical request arrival



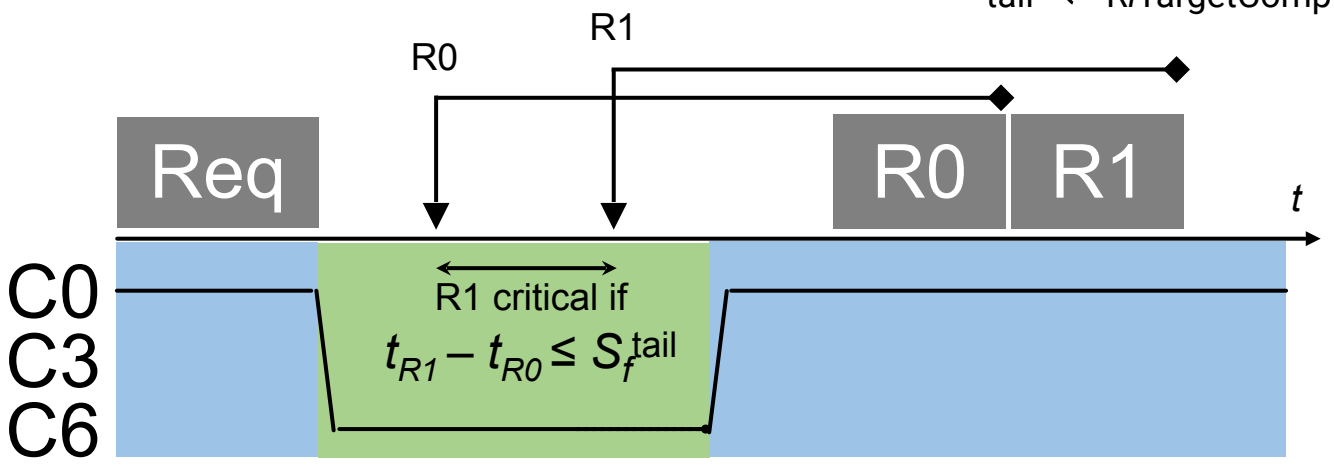
- Detecting critical request arrival
  - If inter-arrival time between 2 consecutive requests are shorter than the tail service time

Coordinate frequency, sleep, delay



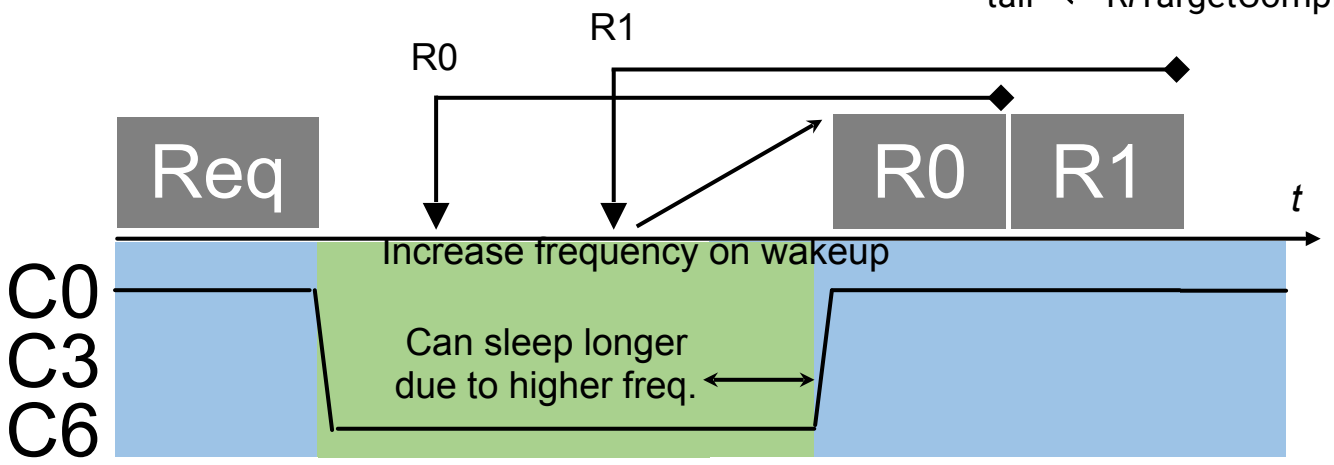
Coordinate frequency, sleep, delay

$f' = S_{tail}/(T_{RiTargetCompletion} - T_{Ri-1Completion})$

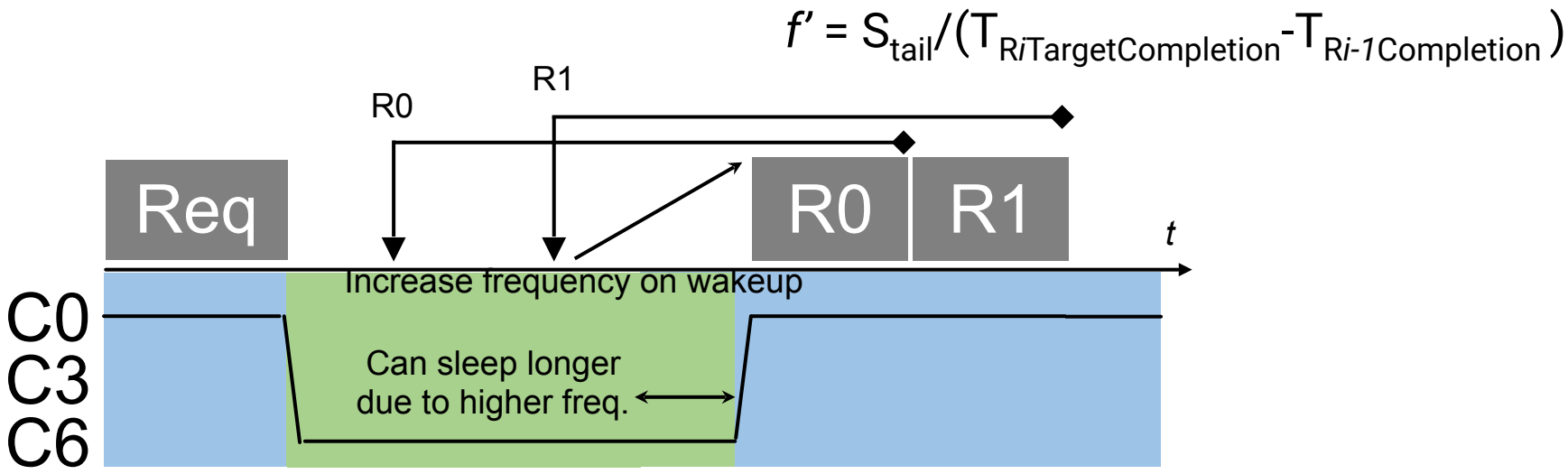


Coordinate frequency, sleep, delay

$$f' = S_{tail}/(T_{Ri\text{TargetCompletion}} - T_{Ri\text{-}1\text{Completion}})$$



Coordinate frequency, sleep, delay



- Reset to lowest frequency on wake up
- Only increase frequency on reconfiguration

Minimize state transition overheads

- Calculate criticality score

$$criticality = \frac{S^{tail} / f}{t_{R_j} - t_{R_{j-1}}}$$

- Send to core that is least critical

$$E(W, f) = \left(W - T_{sleep}\right) P_{idle} + \left(T_{sleep} + T_{wake}\right) P_{max} + T_{dvfs} P_{dvfs} + \left(\frac{S_i}{f}\right) P_f$$

- Minimize state transitions

Algorithm 1: Criticality-aware scheduling

```
1: non_critical_cores = ϕ, non_critical_sleep_cores = ϕ
2: for each core do
3:     compute core_i's criticality
4:     if criticality ≤ 1 then
5:         non_critical_cores ← non_critical_cores ∪ core_i
6:         if core_i is sleeping then
7:             non_critical_sleep_cores ←
               non_critical_sleep_cores ∪ core_i
8:         end if
9:     end if
10: end for
11: if non_critical_cores ≠ ϕ then
12:     if non_critical_sleep_cores ≠ ϕ then
13:         return min(extra energy) in non_critical_sleep_cores
14:     else
15:         return min(extra energy) in non_critical_cores
16:     end if
17: else
18:     return min(extra energy) in all cores
19: end if
```

See paper for details!

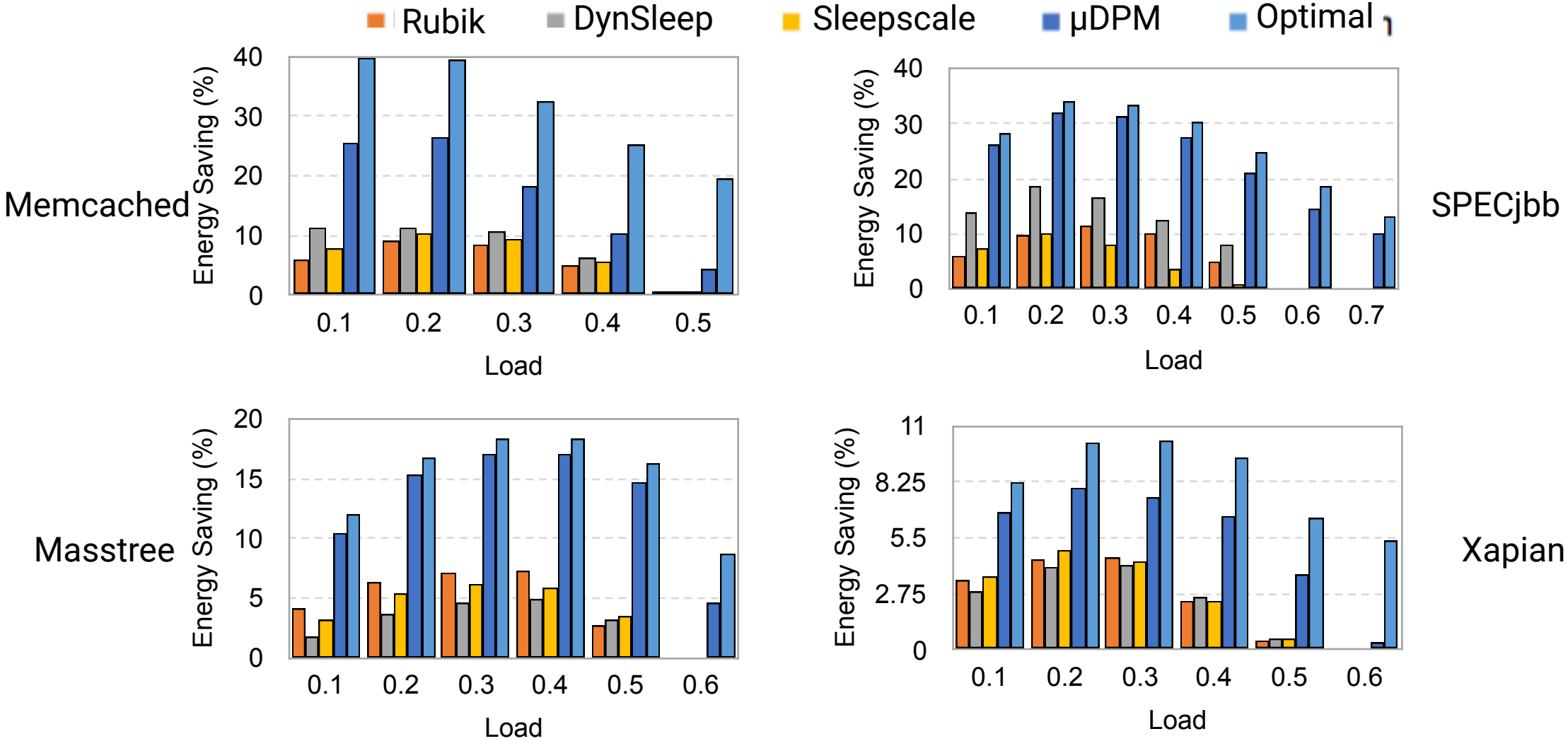
Evaluation

- In-House Simulator (similar to BigHouse)
- Empirical Power Model
  - 10µs DVFS transition,
    89µs sleep transition time (double to account for cache flushing)
  - Add 25µs to first request service time after idle period for cold miss penalty
- Baseline – Linux menu idle governor and intel_pstate driver
- Workloads

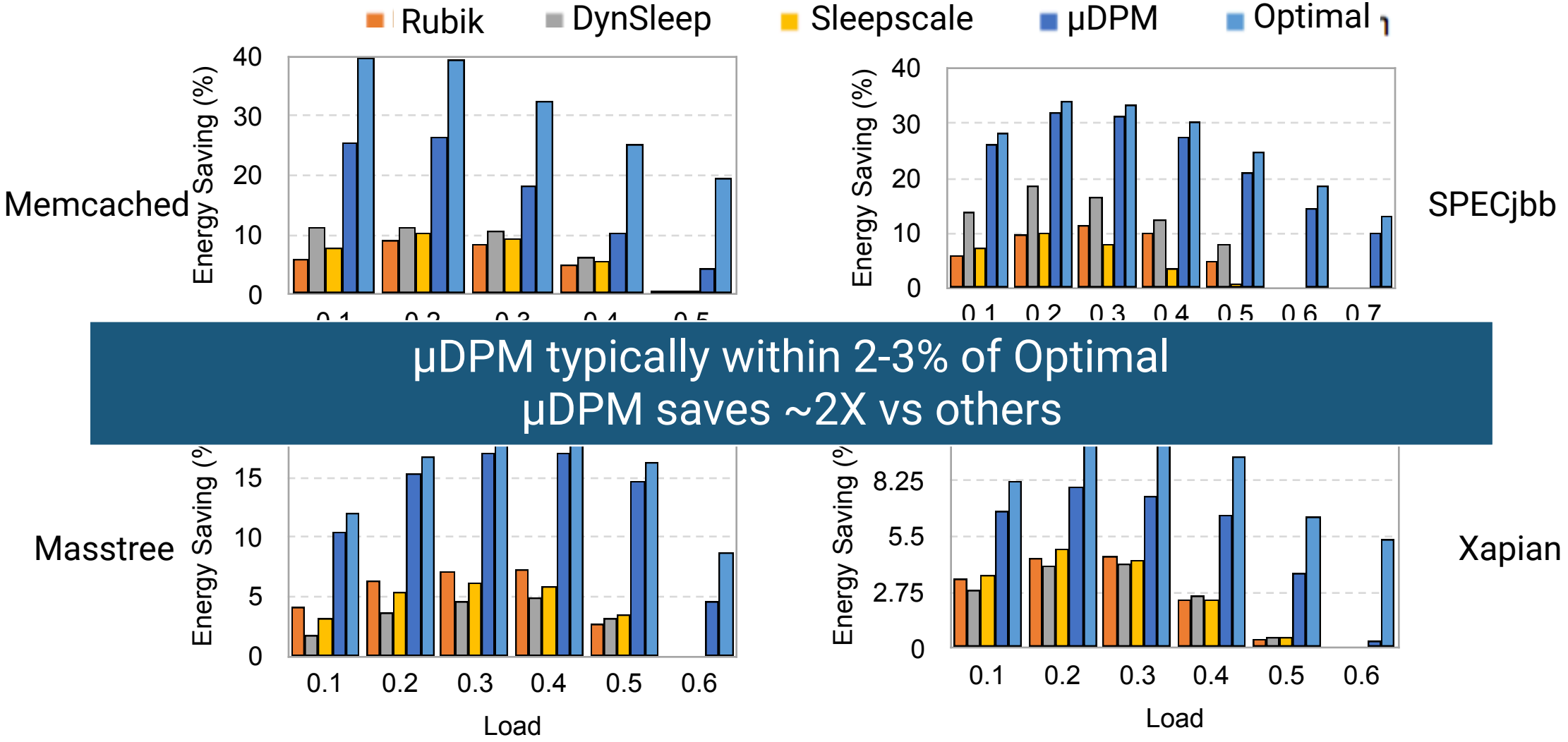
Table II: Workload characteristics.

Name	Avg. Service Time	Tail Service Time	Target Tail Latency
Memcached [43, 44]	30μs	33μs	150μs
SPECjbb [8, 11]	65μs	78μs	800μs
Masstree [8, 11, 46]	246μs	250μs	1100μs
Xapian [8, 11]	431μs	1200μs	2100μs

Energy Savings

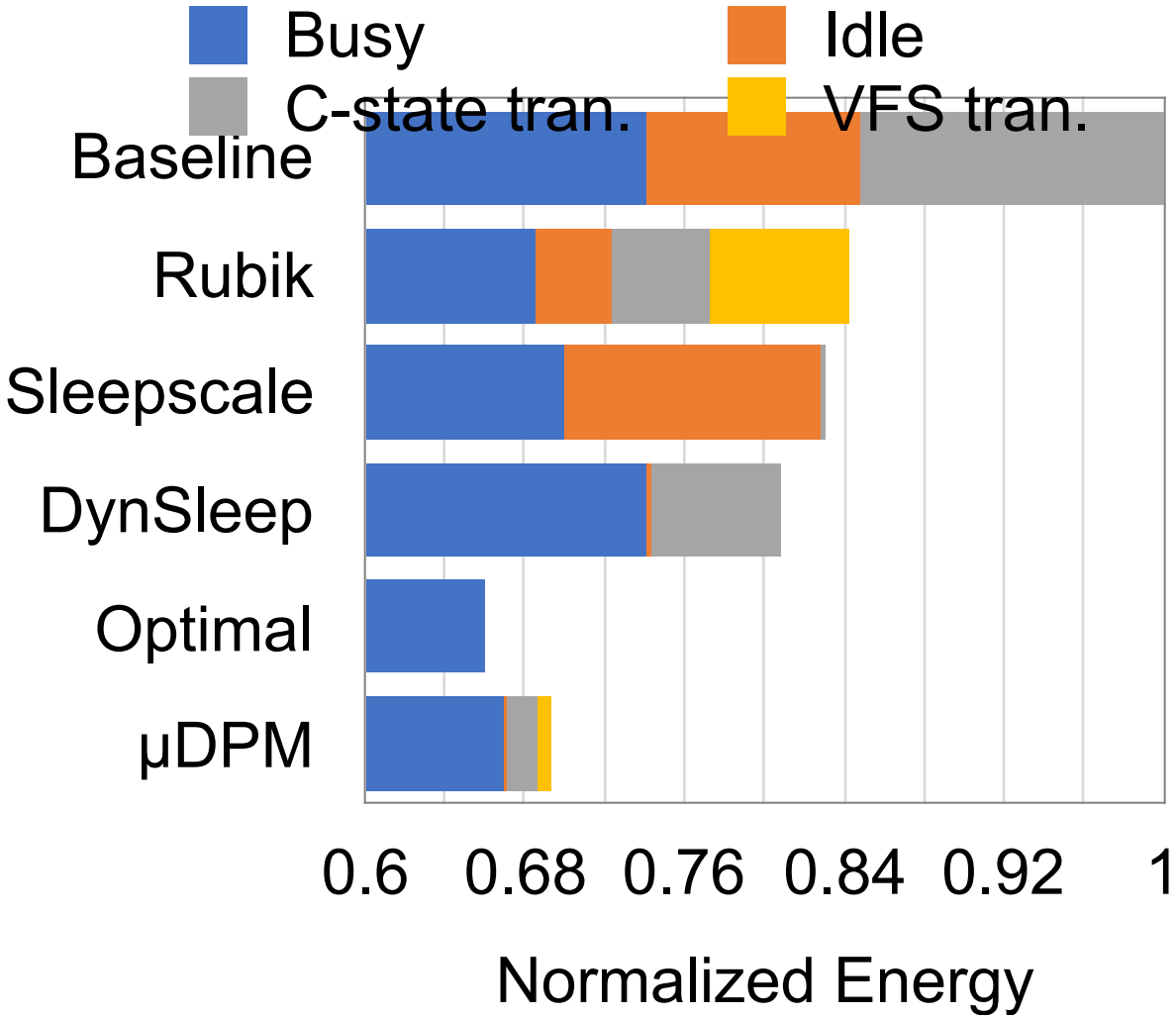


Energy Savings

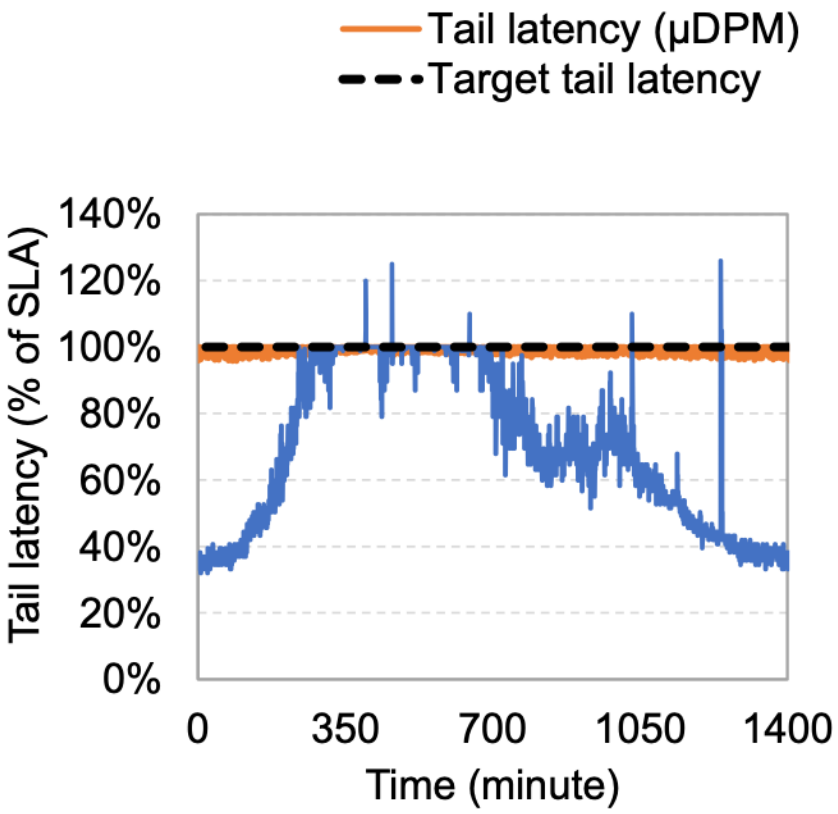


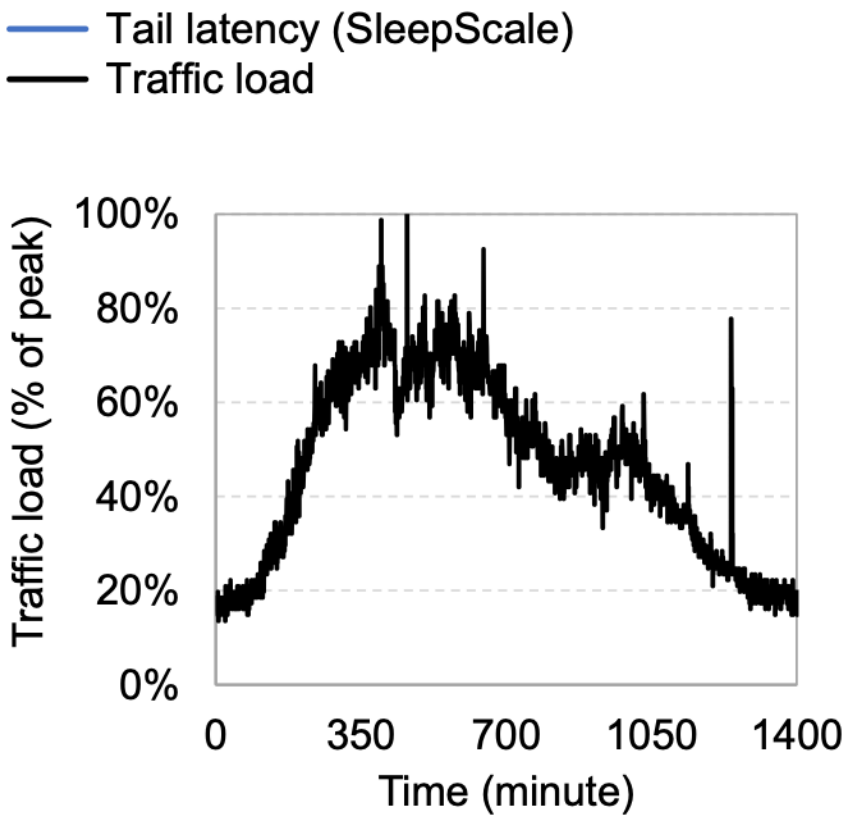
µDPM typically within 2-3% of Optimal
µDPM saves ~2X vs others

State transition overhead reduction



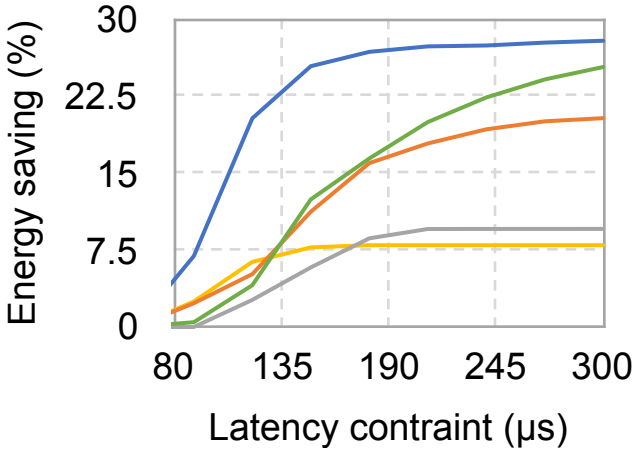
Under Varying Load

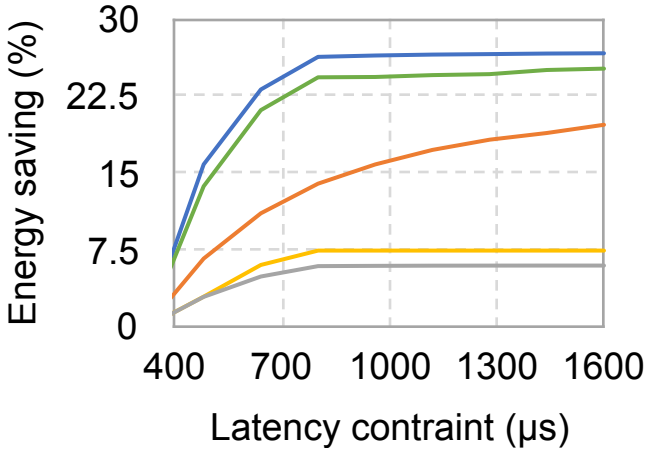




Sensitivity to target tail latency

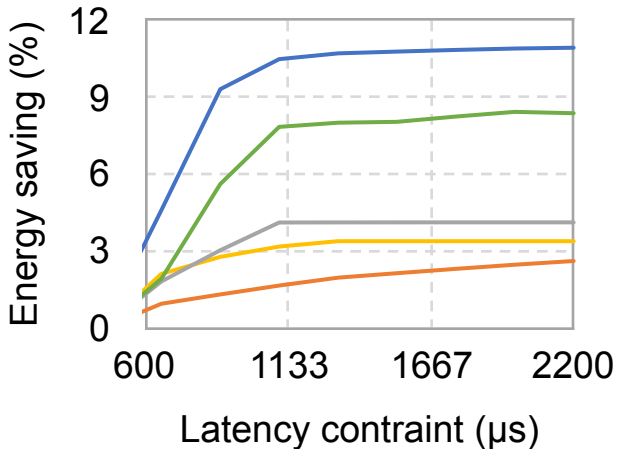
Memcached

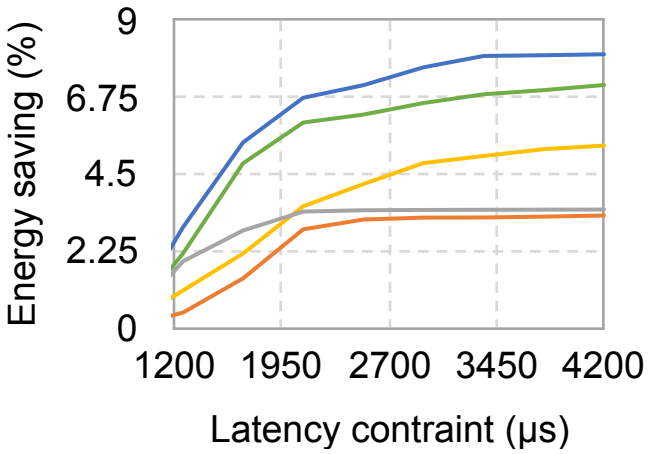




SPECjbb

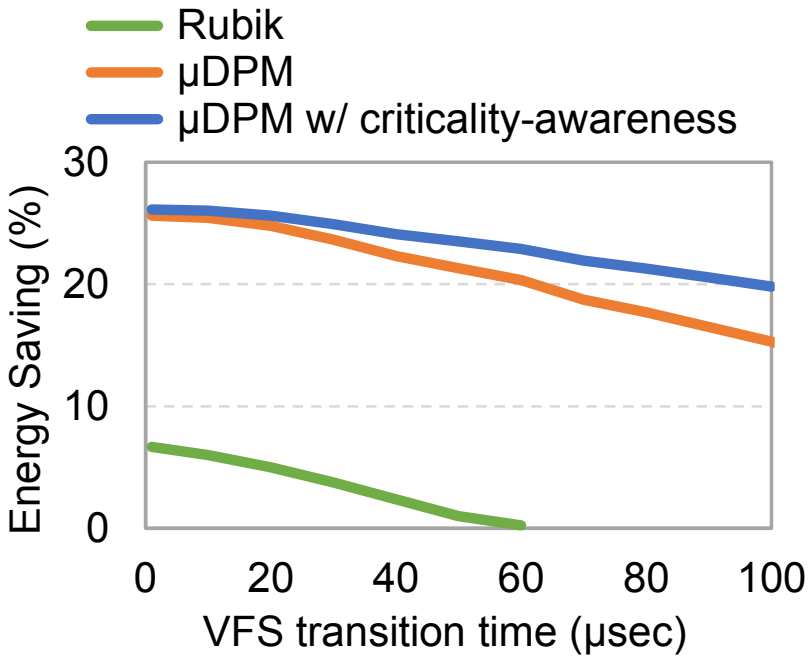
Masstree

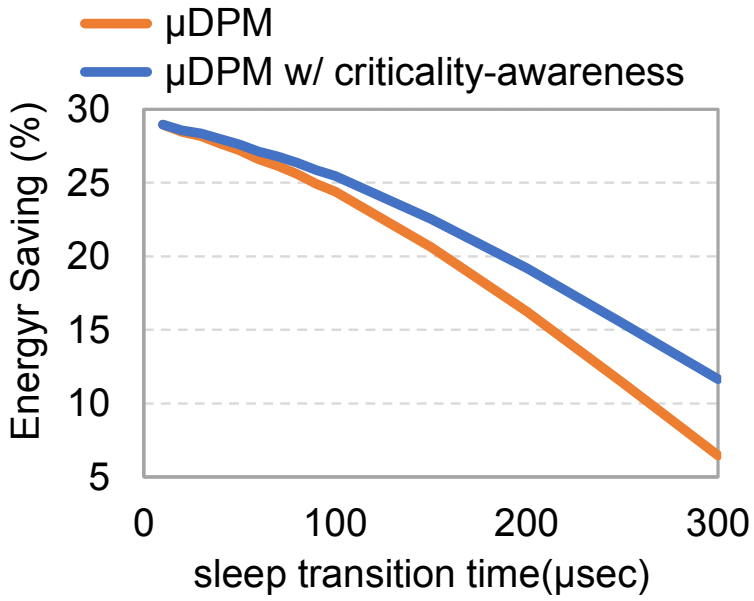




Xapian

Sensitivity to Transition time





Conclusion

- Microsecond *service times* present challenges for Dynamic Power Management
- Careful coordination of DVFS, Sleep and Request delaying can achieve savings with µs service times
- µDPM is able to save ~2x energy compared to state-of-the-art techniques

Thank you!

µDPM: Dynamic Power Management for the Microsecond Era

Chih-Hsun Chou
cchou001@cs.ucr.edu

Laxmi N. Bhuyan
bhuyan@cs.ucr.edu

Daniel Wong
danwong@ucr.edu