On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study

Sathish Gopalakrishnan

The University of British Columbia

(based on work by others at the University of North Carolina)

Focus of this Talk

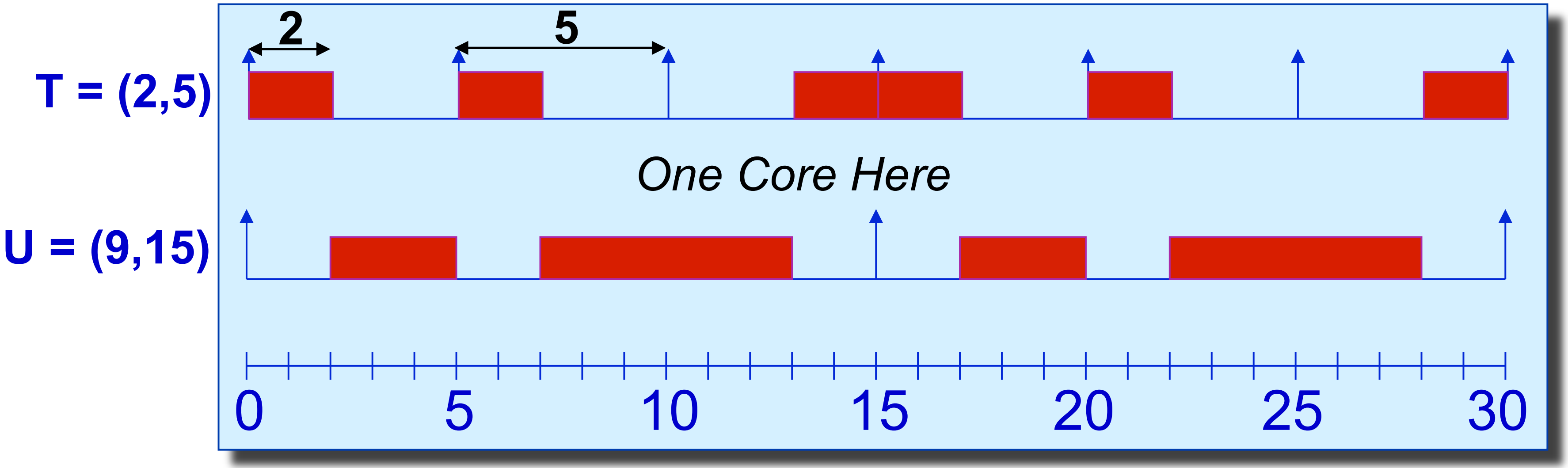
- Multicore platforms are predicted to get much larger in the future.
  - » 10s or 100s of cores per chip, multiple hardware threads per core.
- **Research Question:** How will different real-time scheduling algorithms *scale*?
  - » Scalability is defined w.r.t. *schedulability* (more on this later).

Outline

- Background.
  - Real-time workload assumed.
  - Scheduling algorithms evaluated.
  - Some properties of these algorithms.
- Research questions addressed.
- Experimental results.
- Observations/speculation.
- Future work.

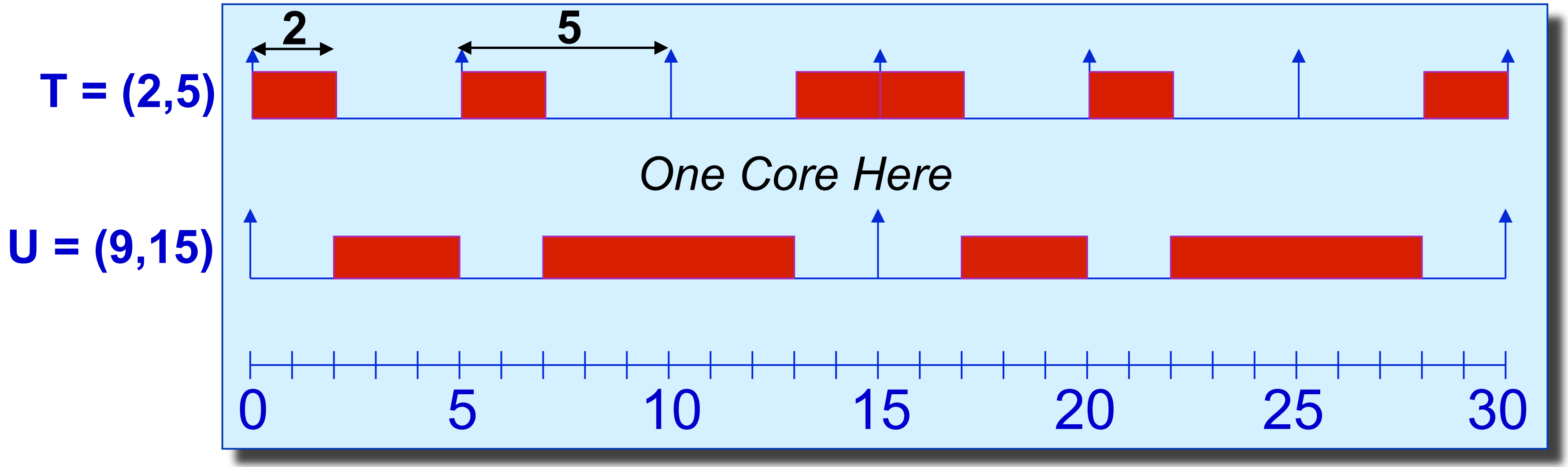
Real-Time Workload Assumed in this Talk

- Set τ of periodic tasks scheduled on M cores:



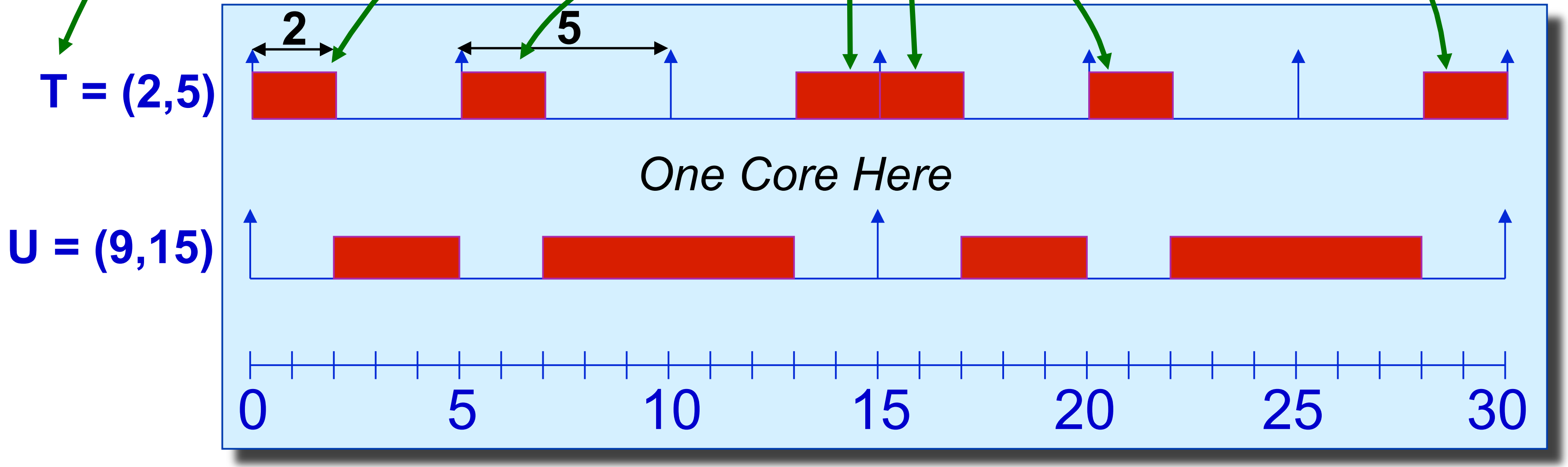
Real-Time Workload Assumed in this Talk

- Set τ of periodic tasks scheduled on M cores:
  - » Task T = (T.e,T.p) releases a *job* with exec. cost T.e every T.p time units.
    - – T's *utilization* (or *weight*) is U(T) = T.e/T.p.
    - – *Total utilization* is $U(\tau) = \Sigma_T$ T.e/T.p.



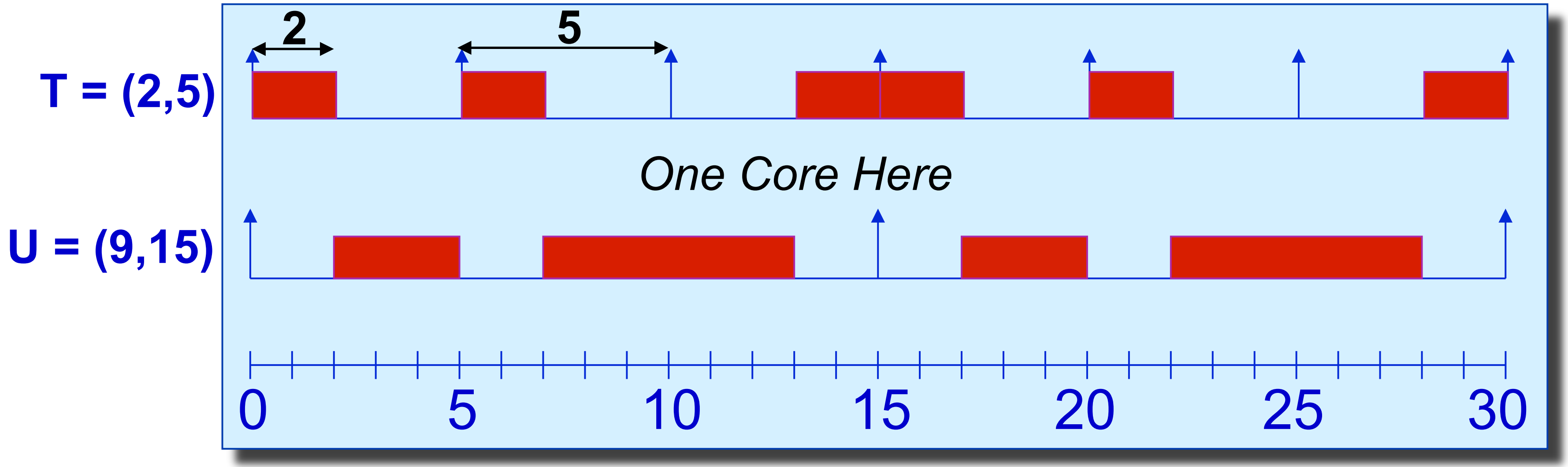
Real-Time Workload Assumed in this Talk

- Set τ of periodic tasks scheduled on M cores:
  - » Task T = (T.e,T.p) releases a *job* with exec. cost T.e every T.p time units.
    - – T's *utilization* (or *weight*) is U(T) = T.e/T.p.
    - – *Total utilization* is $U(\tau) = \Sigma_T T.e/T.p$.



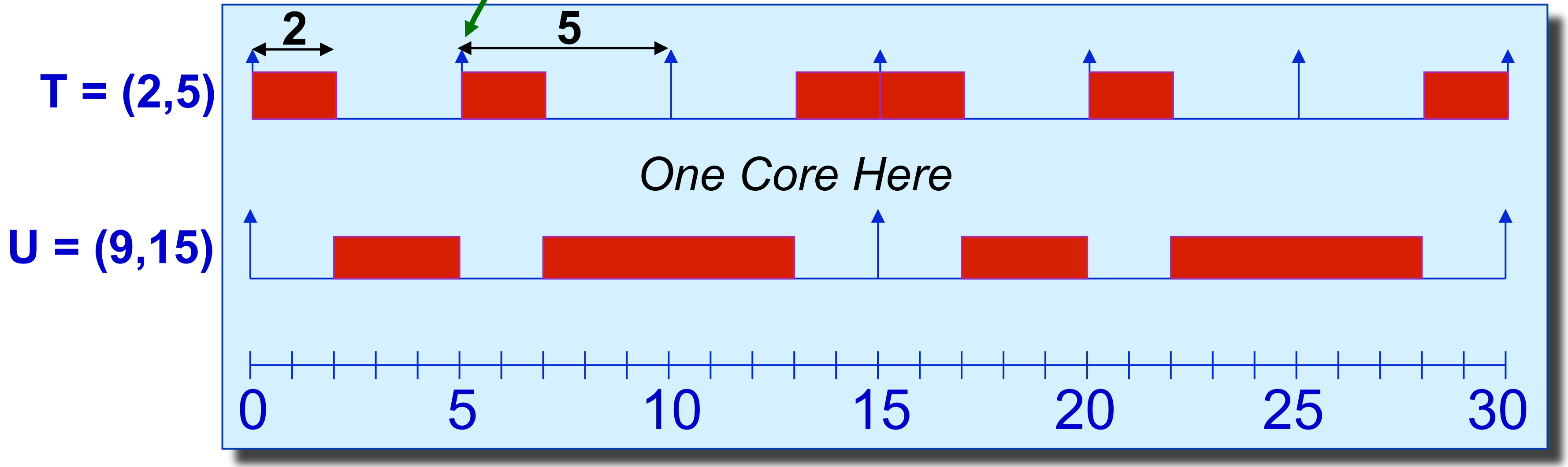
Real-Time Workload Assumed in this Talk

- Set τ of periodic tasks scheduled on M cores:
  - » Task $T = (T.e, T.p)$ releases a *job* with exec. cost $T.e$ every $T.p$ time units.
    - – T's *utilization* (or *weight*) is $U(T) = T.e/T.p$.
    - – *Total utilization* is $U(\tau) = \Sigma_T\ T.e/T.p$.
  - » Each job of T has a *deadline* at the next job release of T.



Real-Time Workload Assumed in this Talk

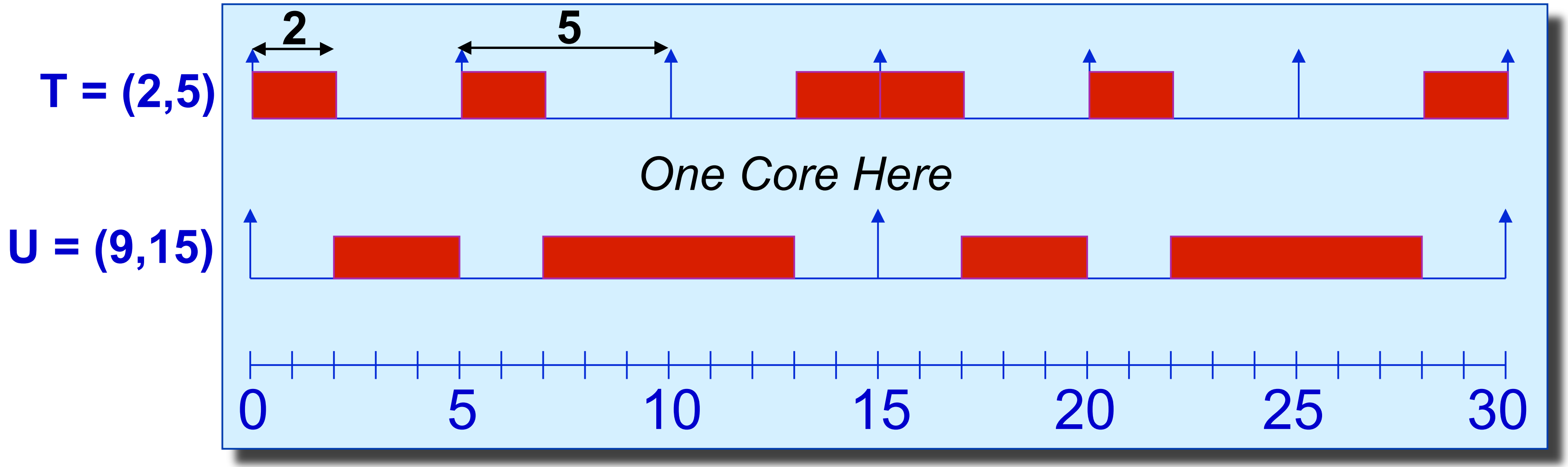
- Set τ of periodic tasks scheduled on M cores:
  - » Task T = (T.e,T.p) releases a *job* with exec. cost T.e every T.p time units.
    - – T's *utilization* (or *weight*) is $U(T) = T.e/T.p$.
    - – *Total utilization* is $U(\tau) = \sum_T T.e/T.p$.
  - » Each job of T has a *deadline* at the next job release of T.



Real-Time Workload Assumed in this Talk

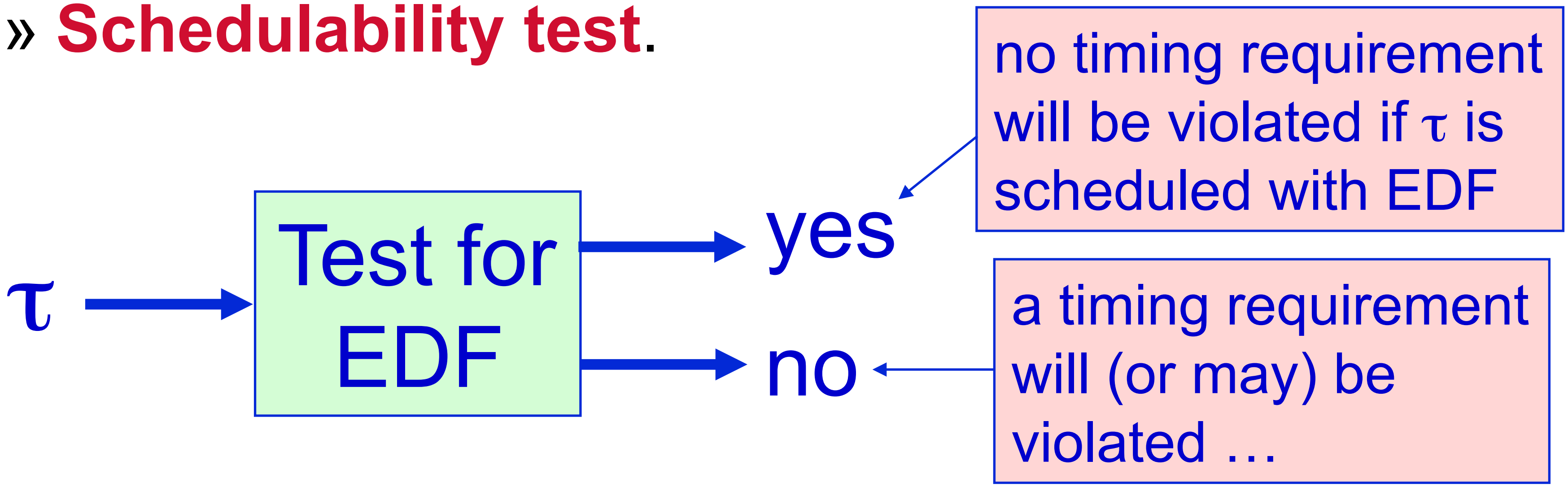
- Set τ of periodic tasks scheduled on M cores:
  - » Task T = (T.e,T.p) releases a *job* with exec. cost T.e every
  - – *Total utilization* is $U(\tau) = \Sigma_T$ T.e/T.p.
  - » Each job of T has a *deadline* at the next job release of T.

This is an *earliest-deadline-first* schedule.
Much of our work pertains to EDF scheduling…



Scheduling vs. Schedulability

- W.r.t. scheduling, we actually care about *two* kinds of algorithms:
  - » **Scheduling algorithm** (of course).
    - – **Example:** Earliest-deadline-first (EDF): Jobs with earlier deadlines have higher priority.
  - » **Schedulability test**.



τ

Test for EDF

yes

no

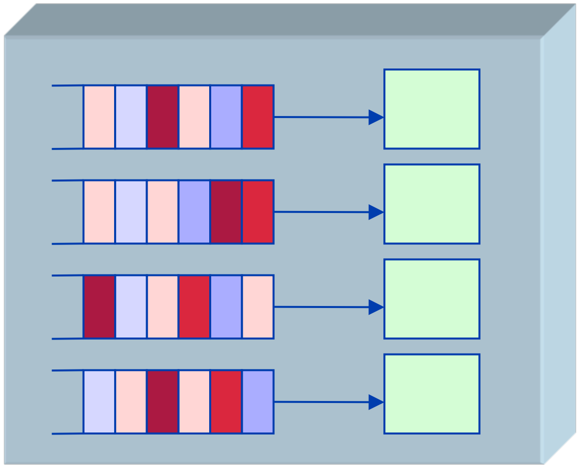
no timing requirement will be violated if τ is scheduled with EDF

a timing requirement will (or may) be violated …

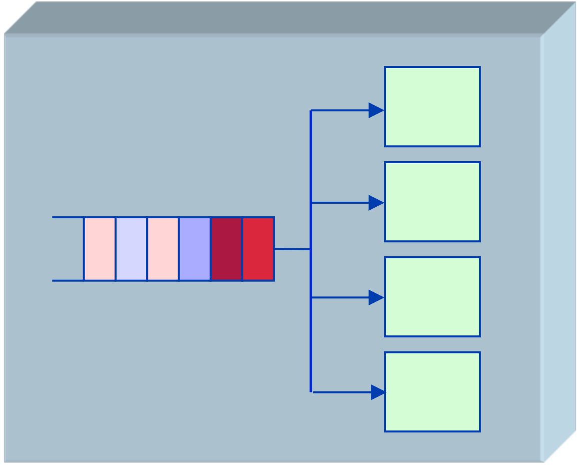
Multiprocessor Real-Time Scheduling

Two Approaches:

Partitioning



Global Scheduling



Steps:

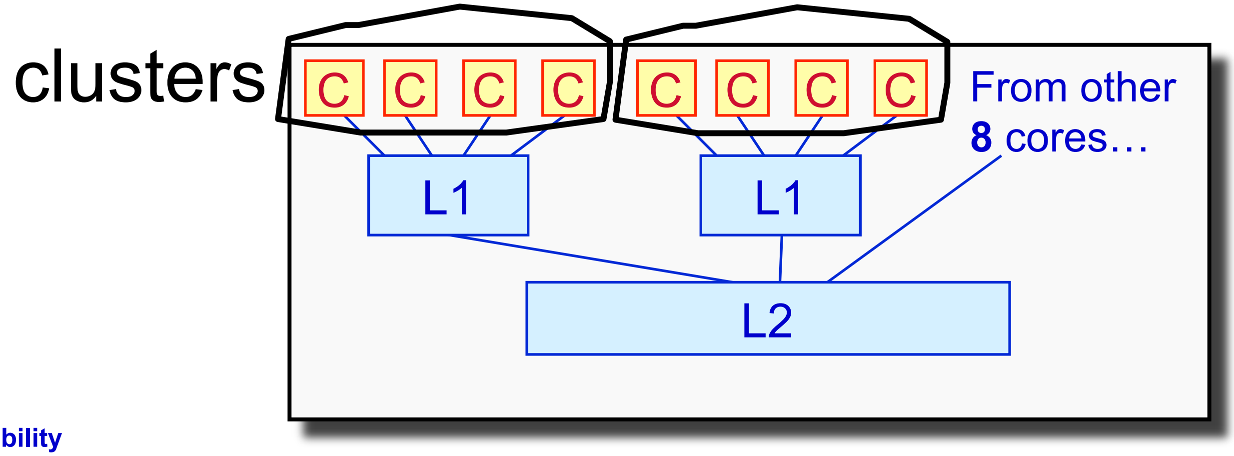
1. Assign tasks to processors (bin packing).
2. Schedule tasks on each processor using a *uniprocessor* algorithm.

Important Differences:

- One task queue.
- Tasks may *migrate* among the processors.

Scheduling Algorithms Considered

- Partitioned EDF: PEDF.
- Preemptive & Non-preemptive Global EDF: GEDF & NP-GEDF.
- Clustered EDF: CEDF.
  - » Partition onto clusters of cores, globally schedule within each cluster

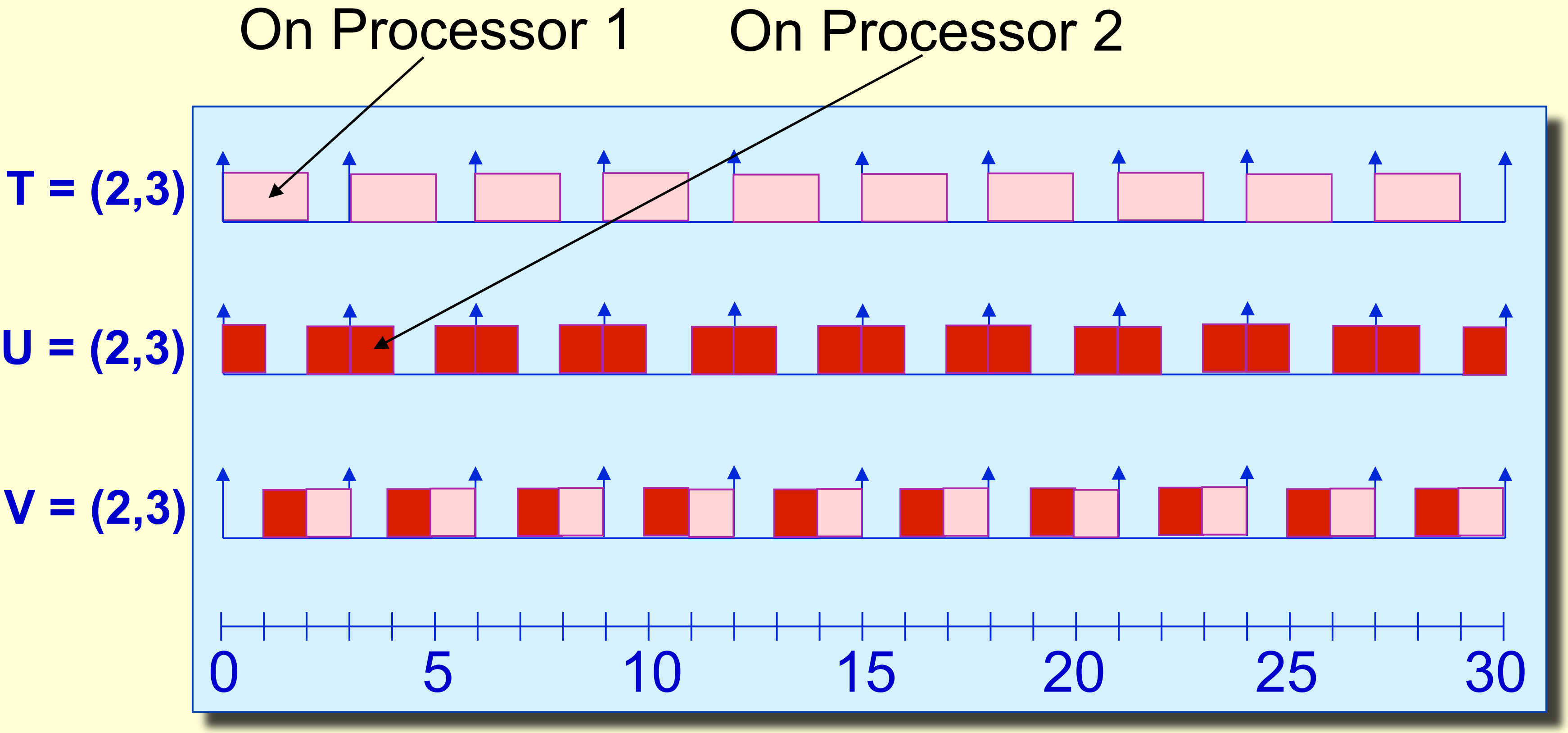


Scheduling Algorithms (Continued)

- PD^2, a global *Pfair* algorithm.
  - » Schedule jobs one quantum at a time at a "uniform" rate.
    - – May preempt and migrate jobs frequently.
- Staggered PD^2: S-PD^2.
  - » Same as PD^2 but quanta are "staggered" to avoid excessive bus contention.

PD^2 Example

3 tasks with parameters (2,3) on two processors…



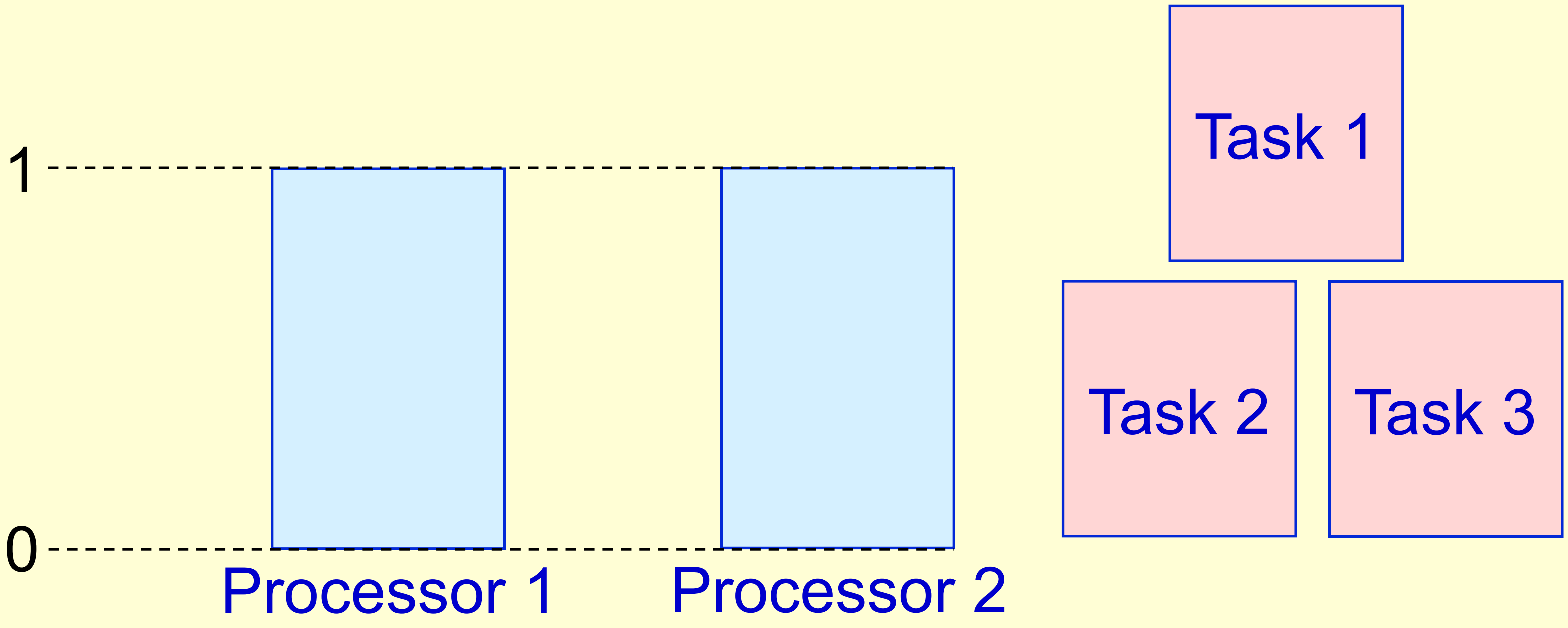
Schedulability

- **HRT:** No deadline is missed.
- **SRT:** Deadline tardiness is bounded.
- For some scheduling algorithms, *utilization loss* is inherent when checking schedulability.
  - » That is, schedulability cannot be guaranteed for all task systems with total utilization at most M.

Example: PEDF

Example: Partitioning three tasks with parameters (2,3) on two processors will *overload* one processor.

In terms of bin-packing…



1

0

Processor 1

Processor 2

Task 1

Task 2

Task 3

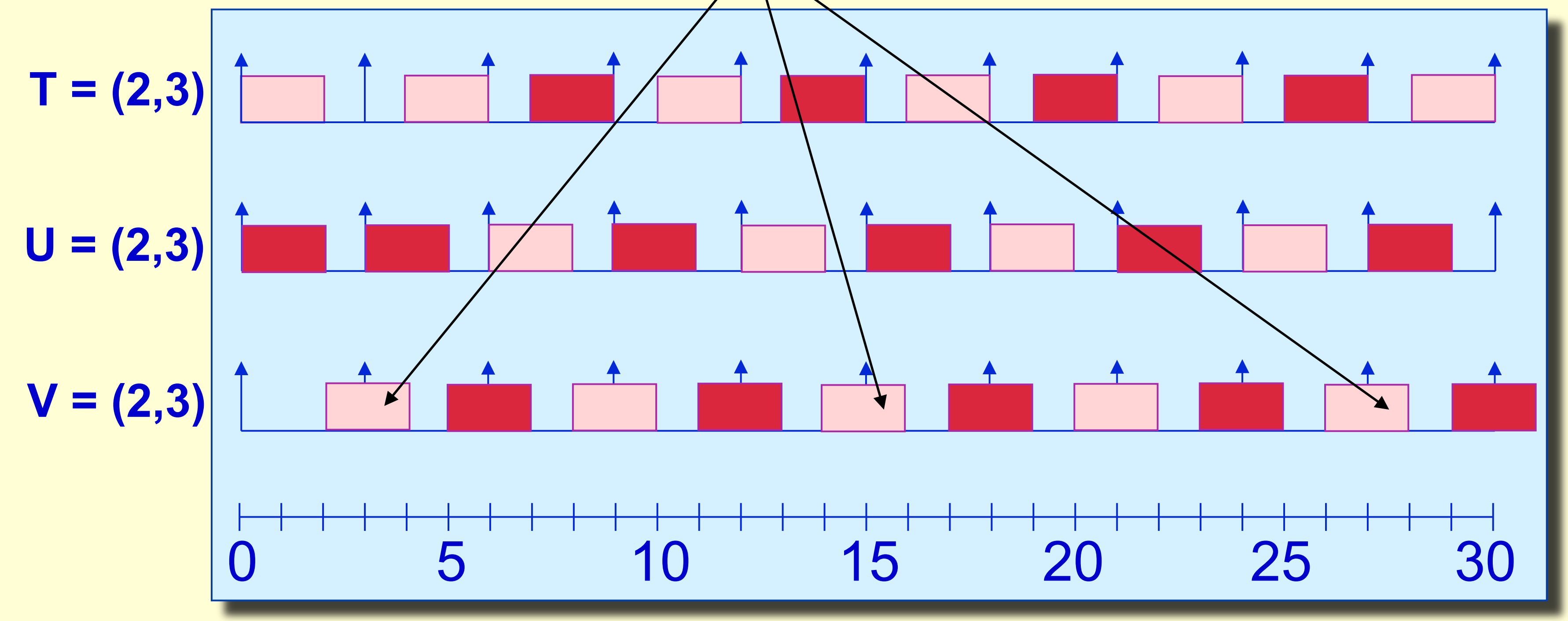
Schedulability Summary

	HRT	SRT
PEDF	util. loss	util. loss (same as HRT)
GEDF	util. loss	no loss
NP-GEDF	util. loss	no loss
CEDF	util. loss	util. loss (not as bad as PEDF)
PD^2	no loss	no loss
$S\text{-}PD^2$	slight loss (must shrink periods by one quantum)	no loss

GEDF SRT Example

Earlier example with GEDF…

Tardiness is at most one quantum.



Outline

- Background.
  - » Real-time workload assumed.
  - » Scheduling algorithms evaluated.
  - » Some properties of these algorithms.
- Research questions addressed.
- Experimental results.
- Observations/speculation.
- Future work.

Research Questions

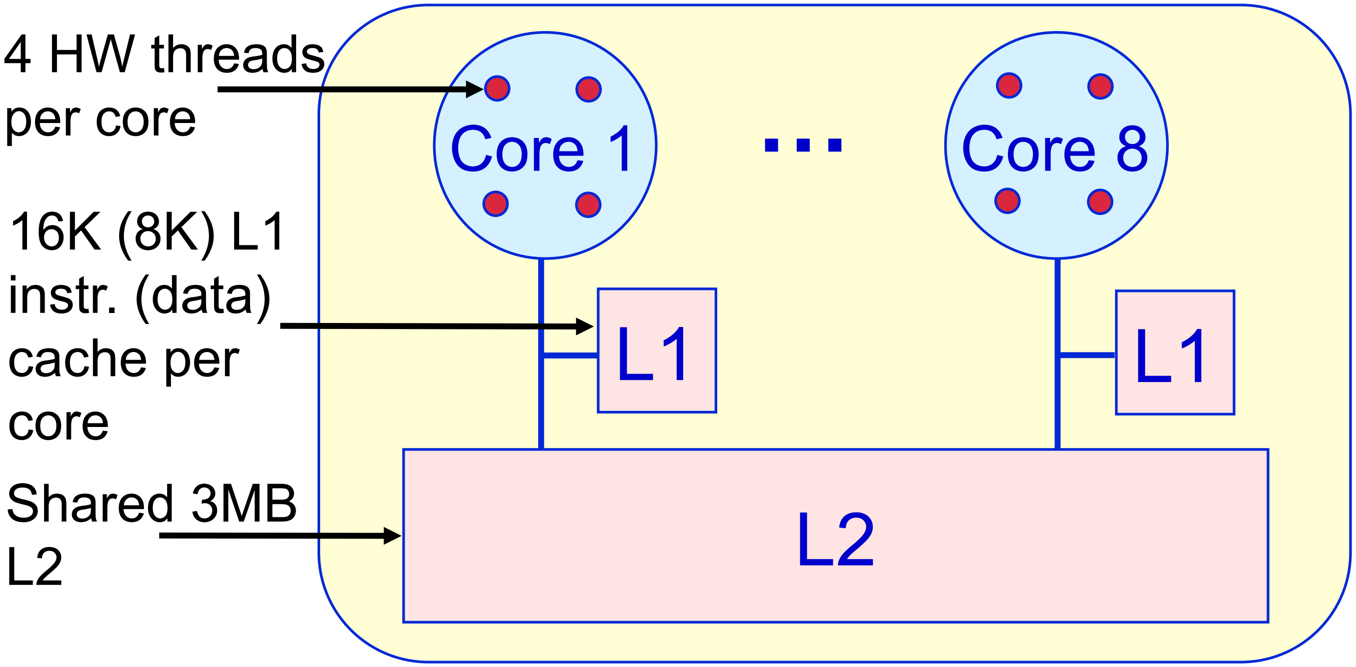
- In *theory*, PD^2 is always preferable.
  - » It is optimal (no utilization loss).

Focus of this Talk: An Experimental comparison of these scheduling algorithms on the basis of *schedulability*.

- Do migrations really matter on a multicore platform with a shared cache?
- As multicore platforms get larger, will global algorithms scale?

Test System

- HW platform: Sun Niagara (UltraSPARC T1).



4 HW threads per core

16K (8K) L1 instr. (data) cache per core

Shared 3MB L2

- 1.2 GHz “RISC-like” cores.
- Relatively simple, e.g., no instr. reordering or branch prediction.
- Caches somewhat small compared to Intel.

- OS has 32 “logical CPUs” to manage.
- ***Far larger than any system considered before in RT literature.***
- **Note:** CEDF “cluster” = 4 HW threads on a core.

Test System (Cont'd)

- Operating System: **LITMUS^{RT}:** LInux Testbed for MUltiprocessor Scheduling in Real-Time systems.
  - » Developed at UNC.
  - » Extends Linux by allowing different schedulers to be linked as "plug-in" components.
  - » Several (real-time) synchronization protocols are also supported.
  - » Code is available at http://www.cs.unc.edu/~anderson/litmus-rt/.

Methodology

- Ran several hundred (synthetic) task sets on the test system.
- Collect nples.
- Distilled SRT) and worst-case (for T) overheads.

Note: This step is *offline*. It does not involve the Niagara.

- Conducted schedulability experiments involving 8.5 million randomly-generated task sets with overheads considered.

Kinds of Overheads

- Tick scheduling overhead.
  - » Incurred when the kernel is invoked at the beginning of
- Re
  - »
- Sc
  - »
- Co
  - » Non-cache-related costs associated with a context switch.
- Preemption/migration overhead.
  - » Costs incurred upon a preemption/migration due to a loss of cache affinity.

These overheads can be accounted for in schedulability tests by inflating job execution costs.

(Doing this correctly is a little tricky.)

Kernel Overheads

- Most overheads were small (2-15μs) except worst-case overheads impacted by global queues.
  - » **Most notable:** Worst-case scheduling overheads for PD^2, S-PD^2, and GEDF/NP-GEDF:

Alg	Scheduling Overhead (in μs)
PD^2	32.7
S-PD^2	43.1
GEDF/NP-GEDF	55.2+.26N (N = no. of tasks)

Preemption/Migration Overheads

- Obtained by measuring synthetic tasks, each with a 64K working set & 75/25 read/write ratio.
  - » **Interesting trends:** PD^2 is terrible, staggering really helps, preempt. cost ≈ mig. cost per algorithm, but algorithms that migrate have higher costs.

***Worst-Case* Overheads (in μs)**

Alg	Overall	Preemption	Intra-Cluster Mig	Inter-Cluster Mig
PD^2	681.1	649.4	654.2	681.1
S-PD^2	104.1	103.4	103.4	104.1
GEDF	375.4	375.4	326.8	321.1
CEDF	171.6	171.6	167.3	---
PEDF	139.1	139.1	---	---

Schedulability Results

- Generated random tasks using 6 distributions and checked schedulability using “state-of-the-art” tests (with overheads considered).
  - » 8.5 million task sets in total.
- Distributions:
  - » Utilizations **uniform** over
    - – [0.001,01] (**light**),
    - – [0.1,0.4] (**medium**), and
    - – [0.5,09] (**heavy**).
  - » **Bimodal** with utilizations distributed over either [0.001,05) or [0.5,09] with probabilities of
    - – 8/9 and 1/9 (**light**),
    - – 6/9 and 3/9 (**medium**), and
    - – 4/9 and 5/9 (**heavy**).

Schedulability Results

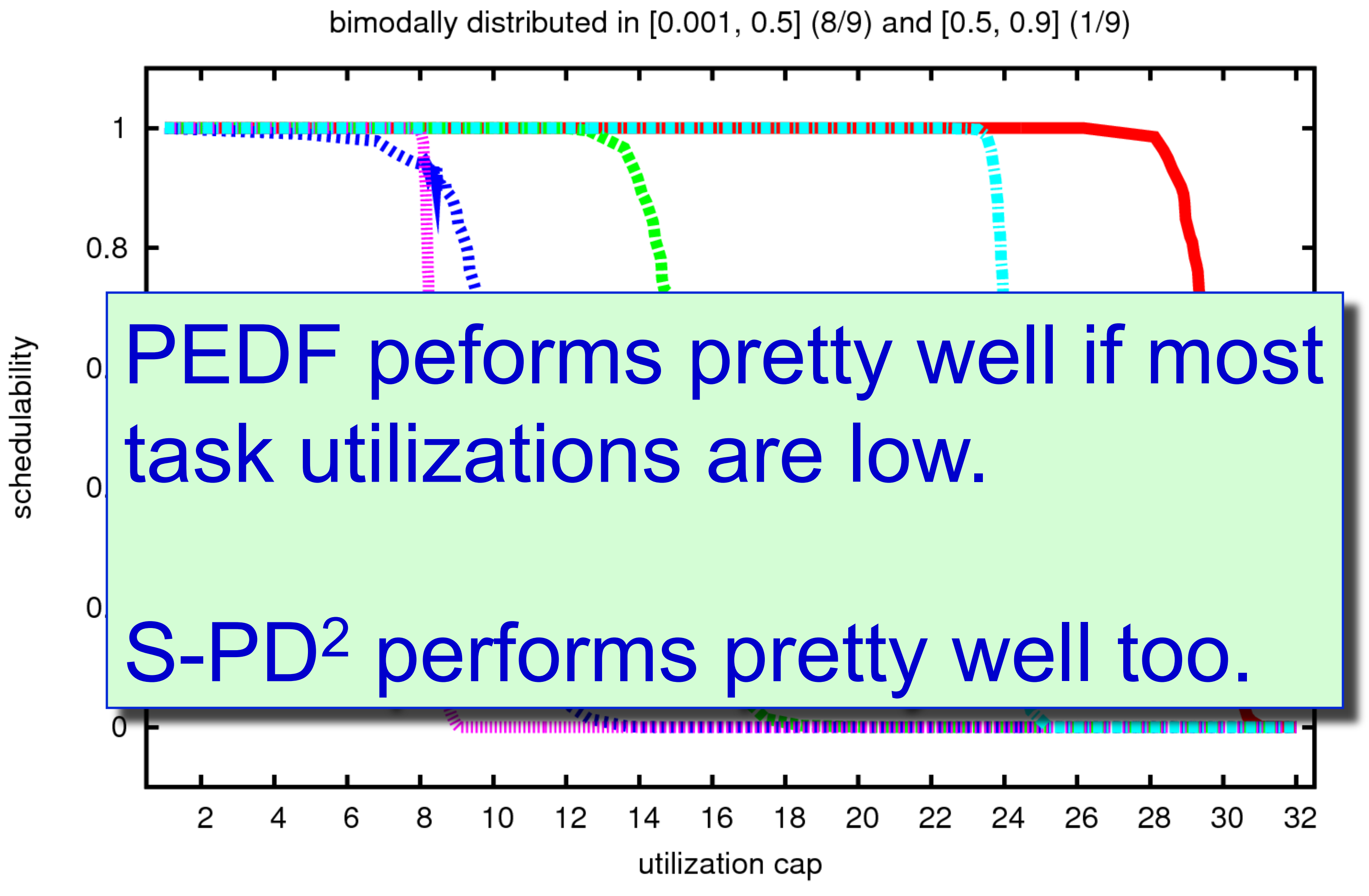
- Generated random tasks using 6 distributions and checked schedulability using “state-of-the-art” tests (with overheads considered).
  - » 8.5 million task sets in total.
- Distributions:
  - » Utilizations **uniform** over
    - – [0.001,01] (**light**),
    - – [0.1,0.4] (**medium**), and
    - – [0.5,09] (**heavy**).
  - » **Bimodal** with utilizations distributed over either [0.001,05) or [0.5,09] with probabilities of
    - – 8/9 and 1/9 (**light**),
    - – 6/9 and 3/9 (**medium**), and
    - – 4/9 and 5/9 (**heavy**).

will only show graphs for these

HRT Summary

- PEDF usually wins.
  - Exception: Lots of heavy tasks (makes bin-packing hard).
- S-PD^2 usually does well.
  - Staggering has an impact.
- PD^2 and GEDF are quite poor.
  - PD^2 is negatively impacted by high preemption and migration costs due to aligned quanta.
  - GEDF suffers from high scheduling costs (due to the global queue).

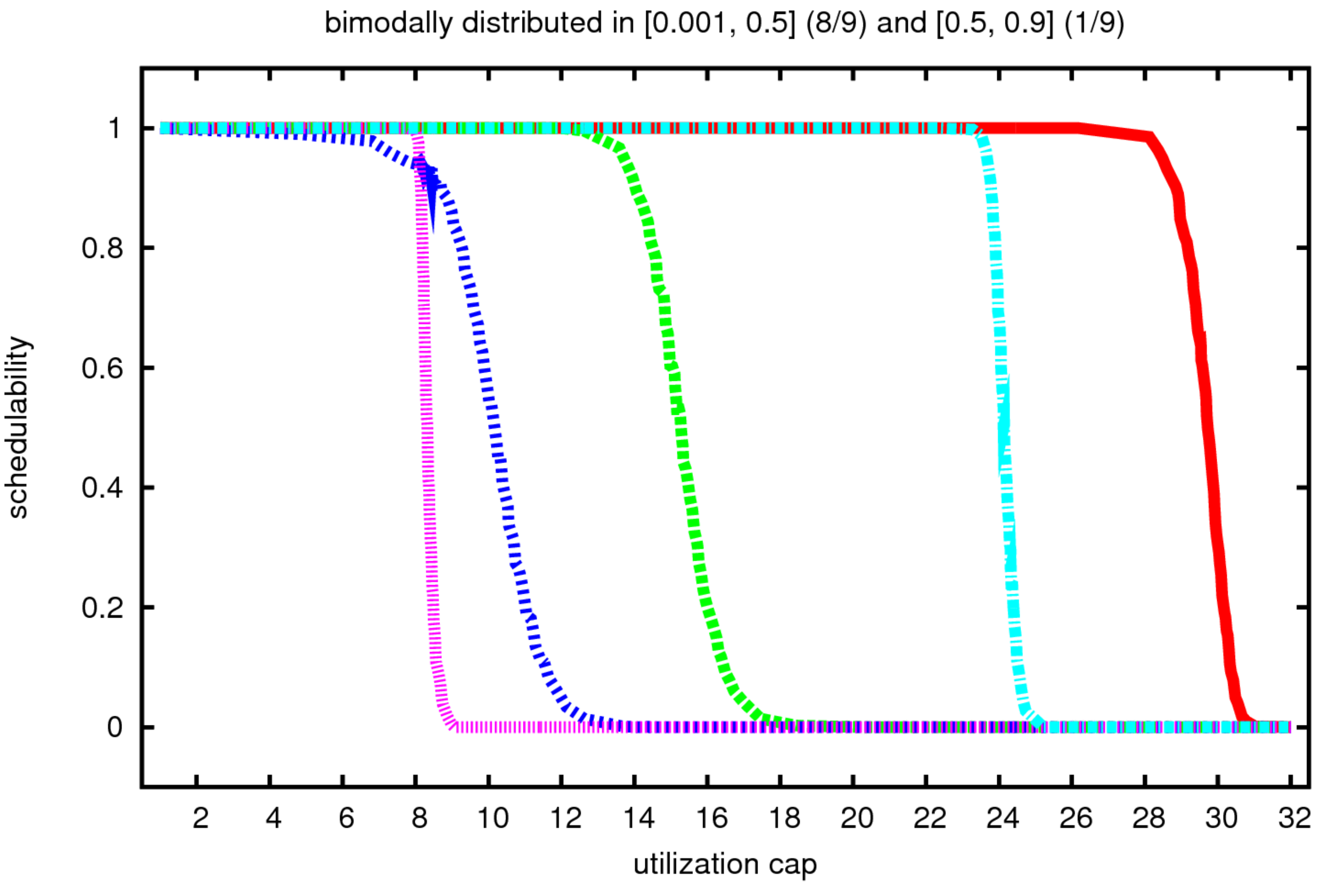
HRT, Bimodal Light



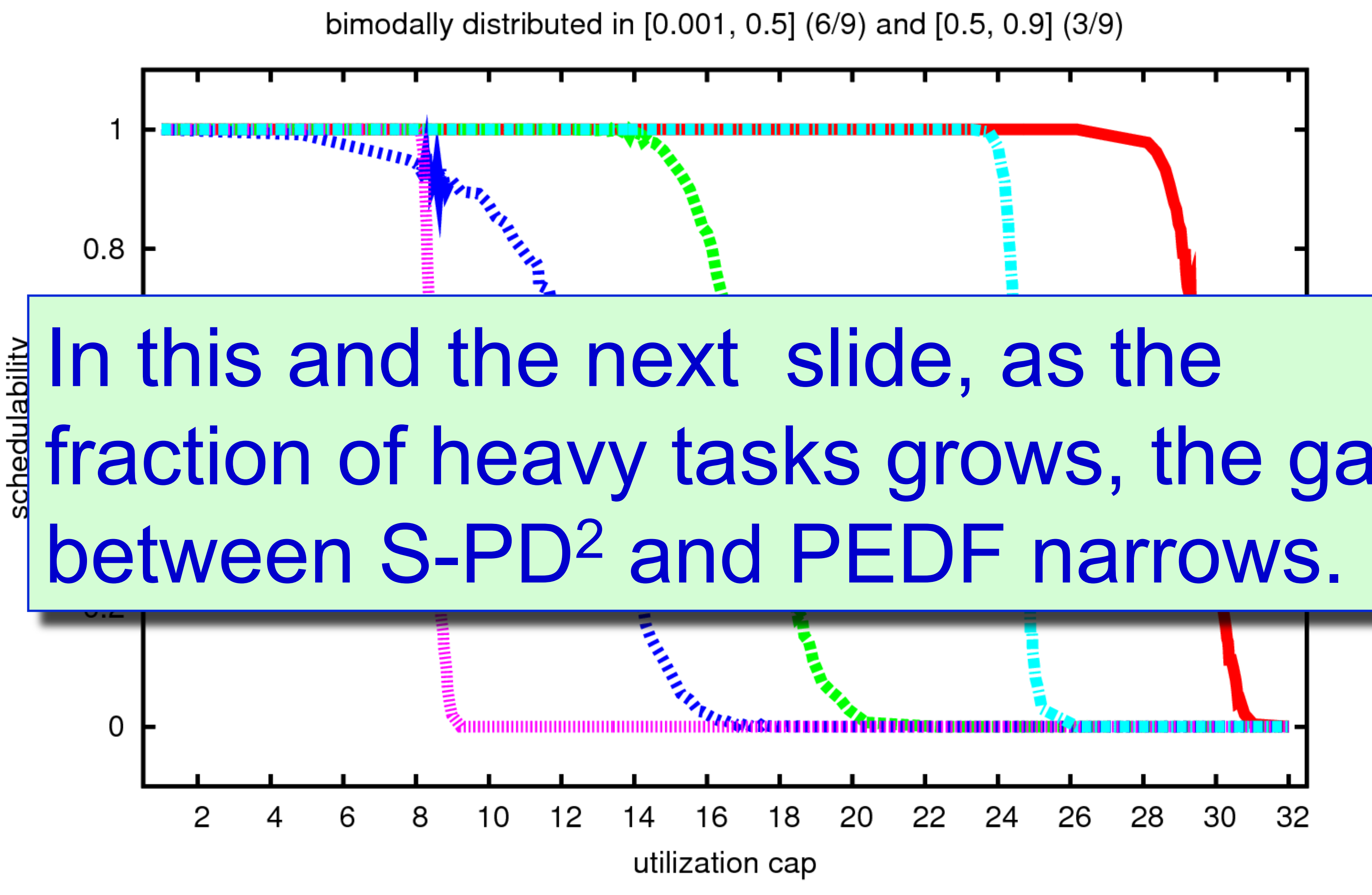
PEDF peforms pretty well if most task utilizations are low.

S-PD^2 performs pretty well too.

HRT, Bimodal Light

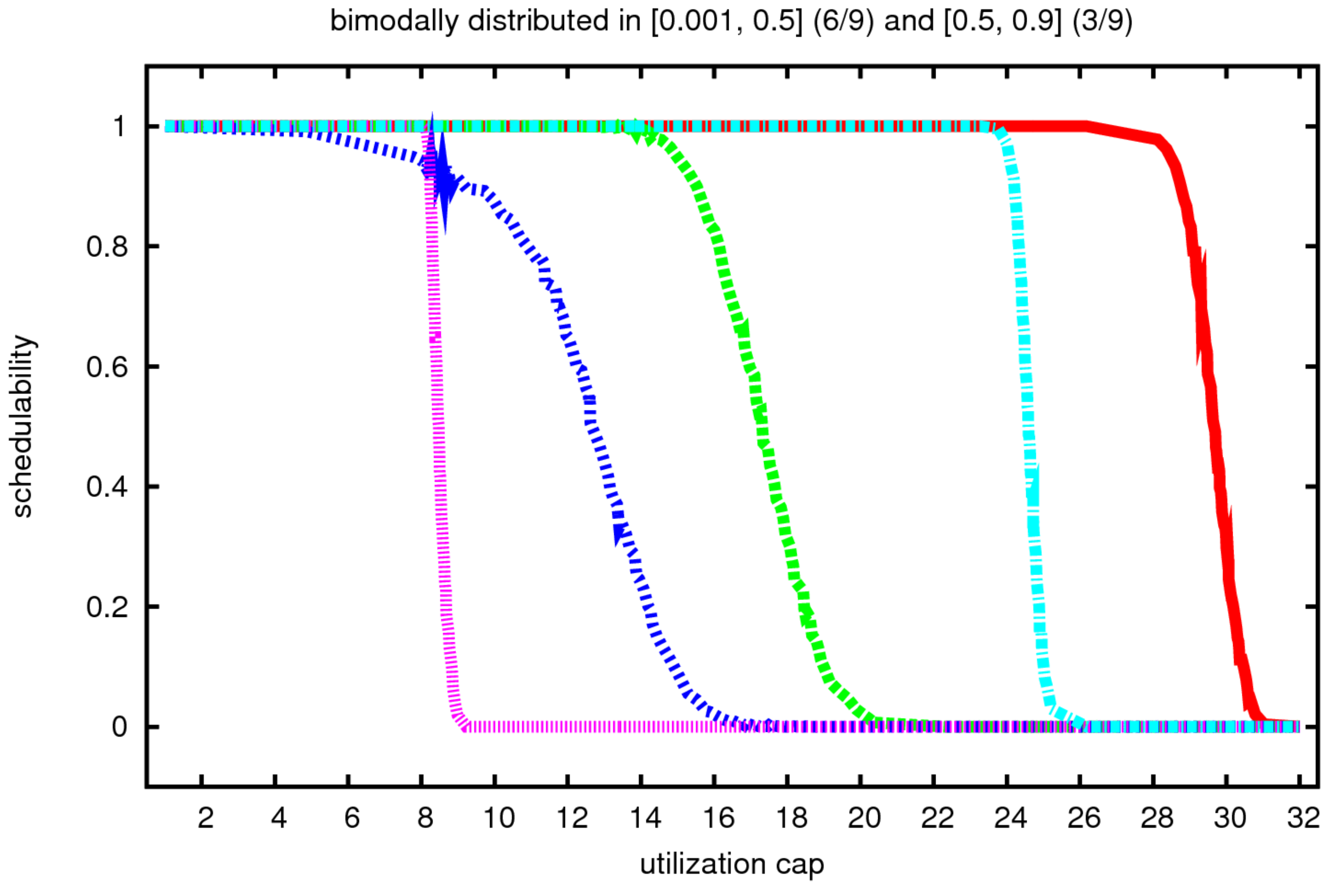


HRT, Bimodal Medium

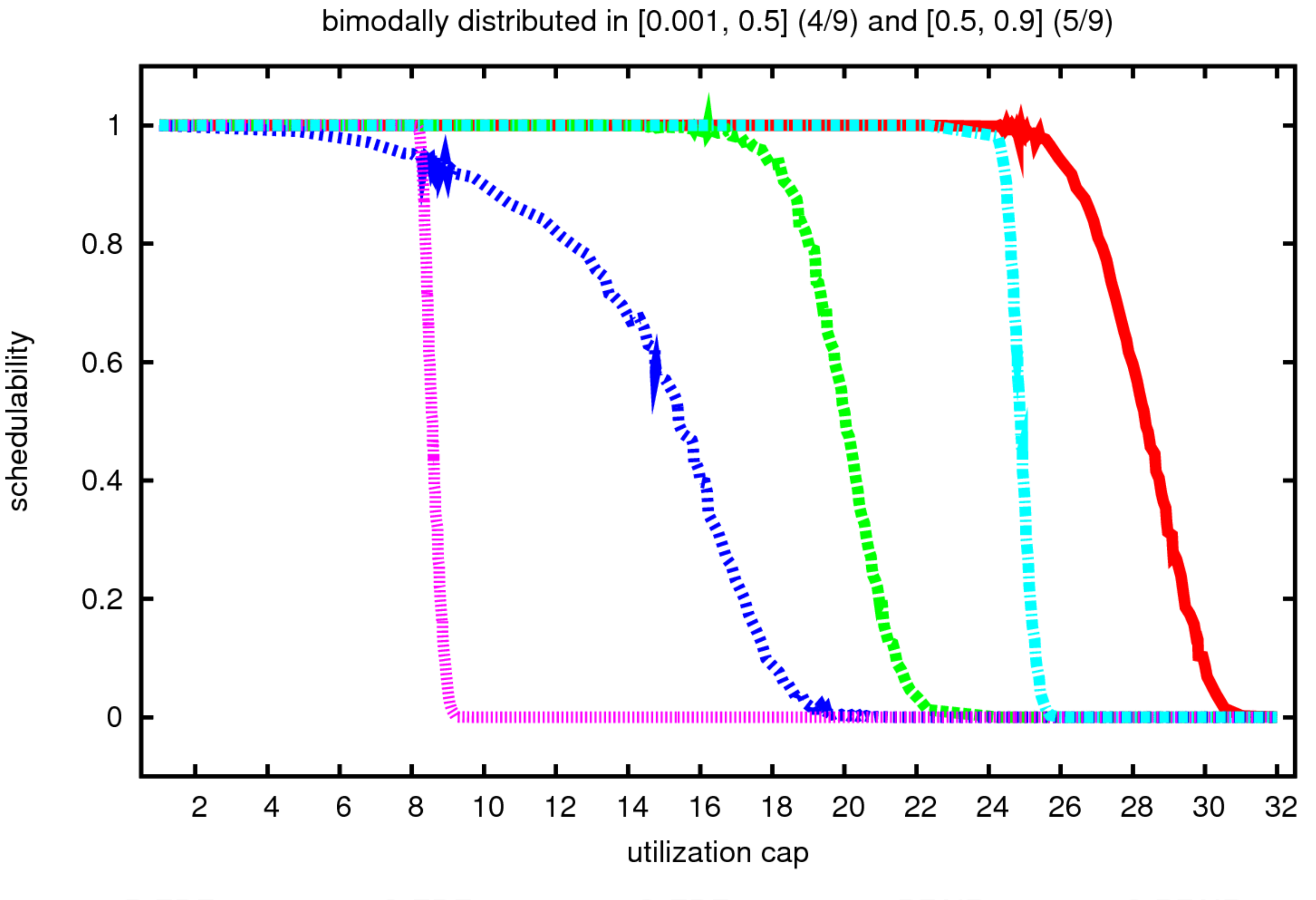


In this and the next slide, as the fraction of heavy tasks grows, the gap between S-PD^2 and PEDF narrows.

HRT, Bimodal Medium



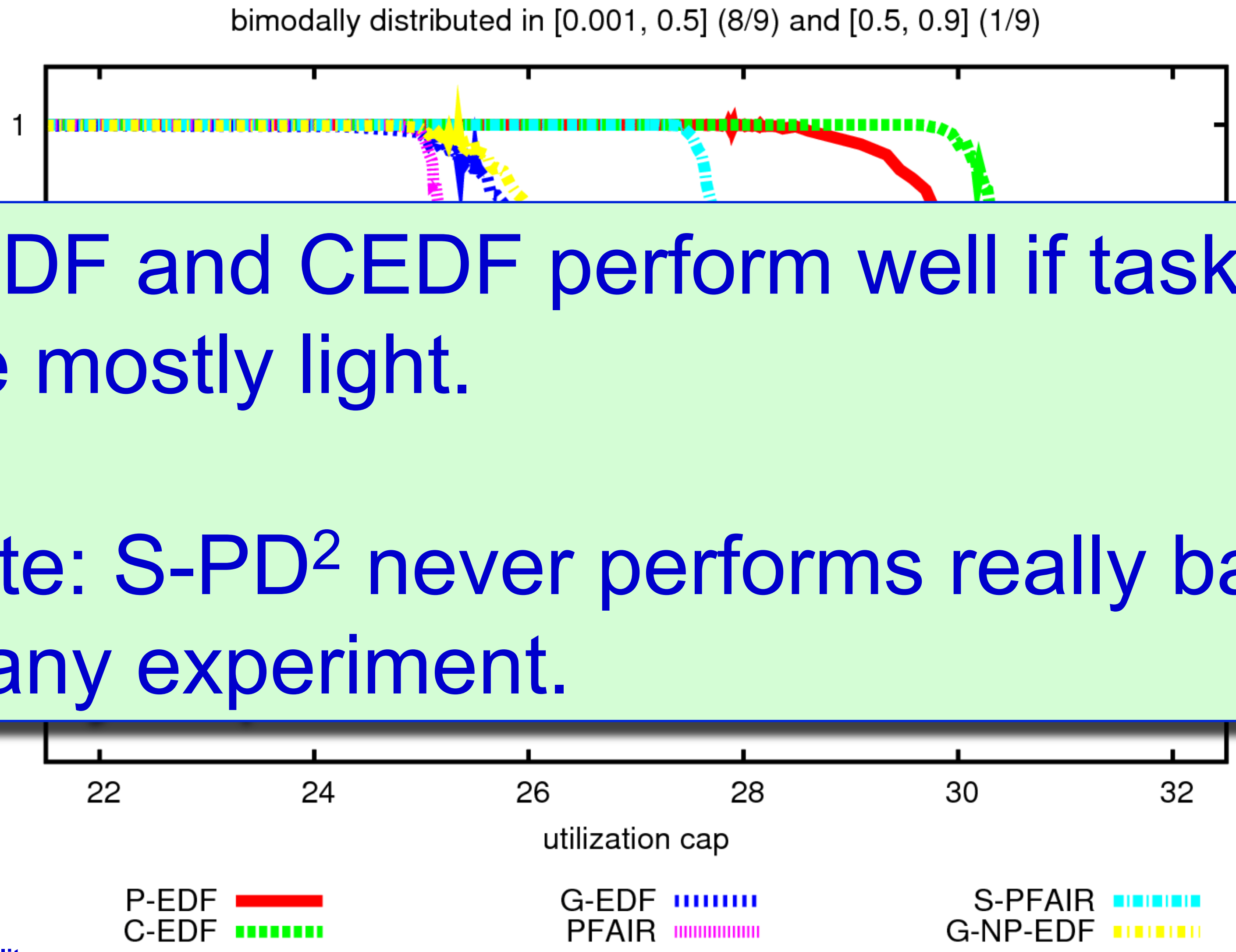
HRT, Bimodal Heavy



SRT Summary

- PEDF is not as effective as before, but still OK in light-mostly cases.
- CEDF performs the best in most cases.
- S-PD^2 still performs generally well.
- GEDF is still negatively impacted by higher scheduling costs.
  - » Note: SRT schedulability for GEDF entails no utilization loss.
  - » NP-GEDF and GEDF are about the same.
- Note: The scale is different from before.

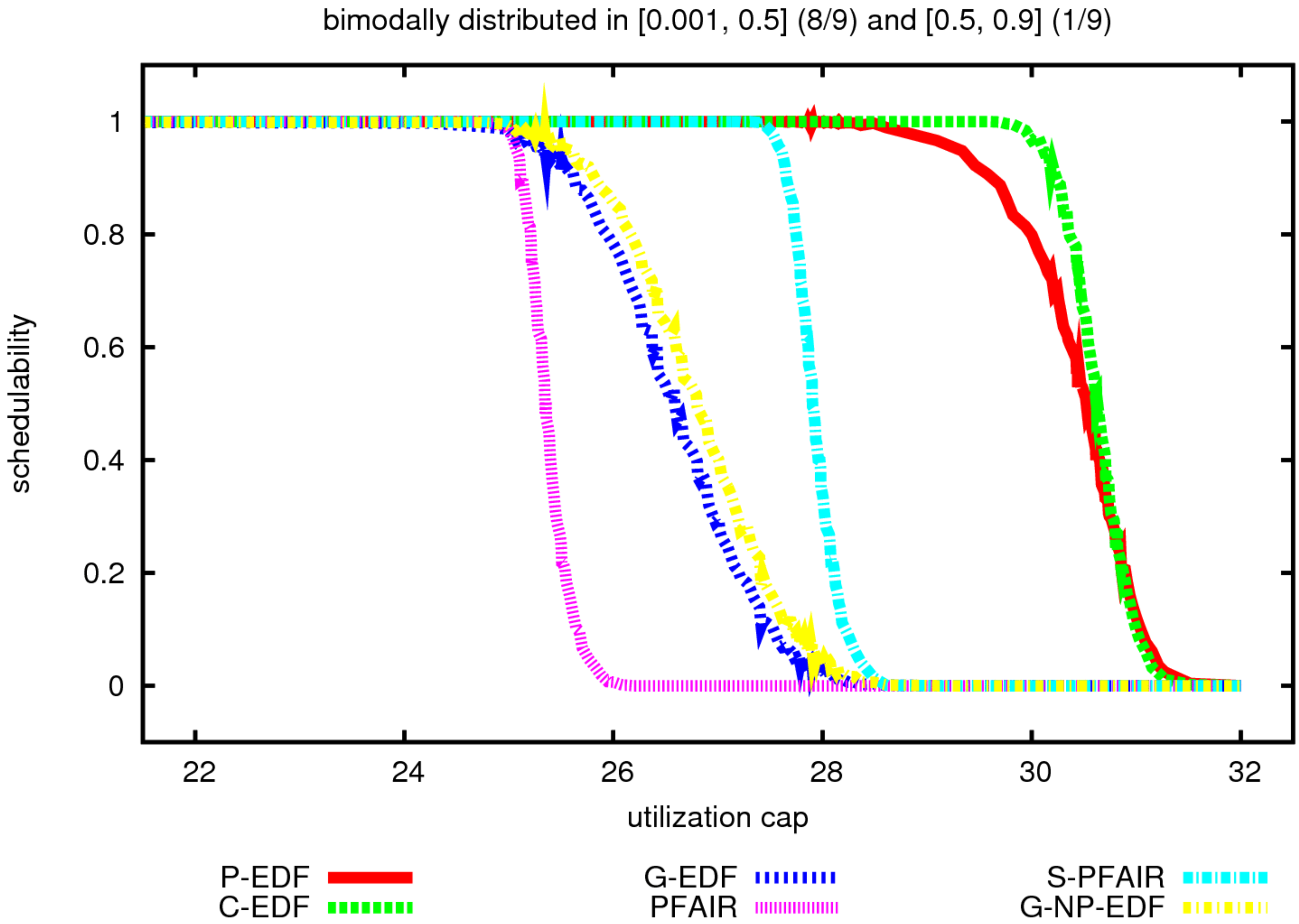
SRT, Bimodal Light



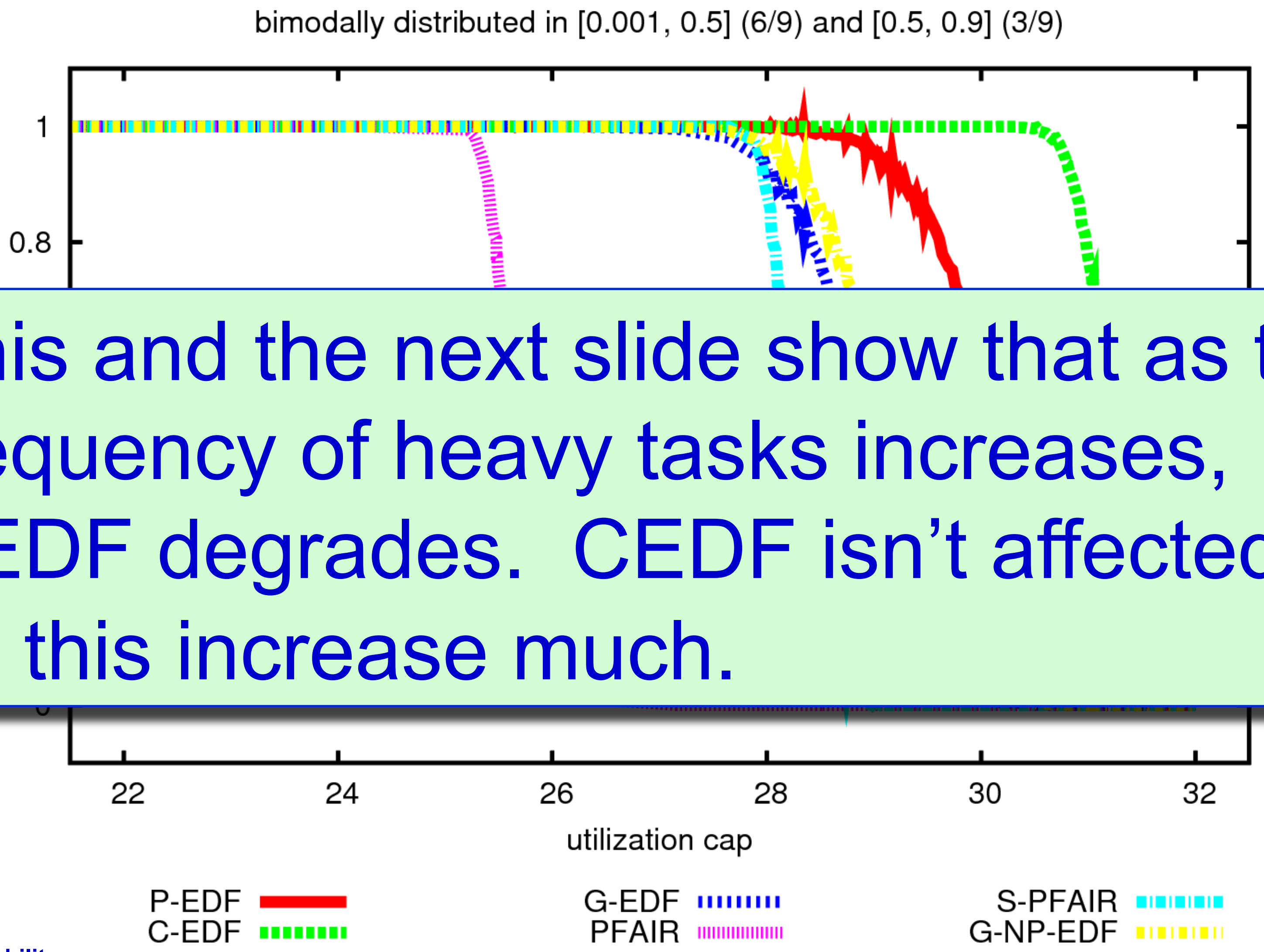
PEDF and CEDF perform well if tasks are mostly light.

Note: S-PD^2 never performs really badly in any experiment.

SRT, Bimodal Light

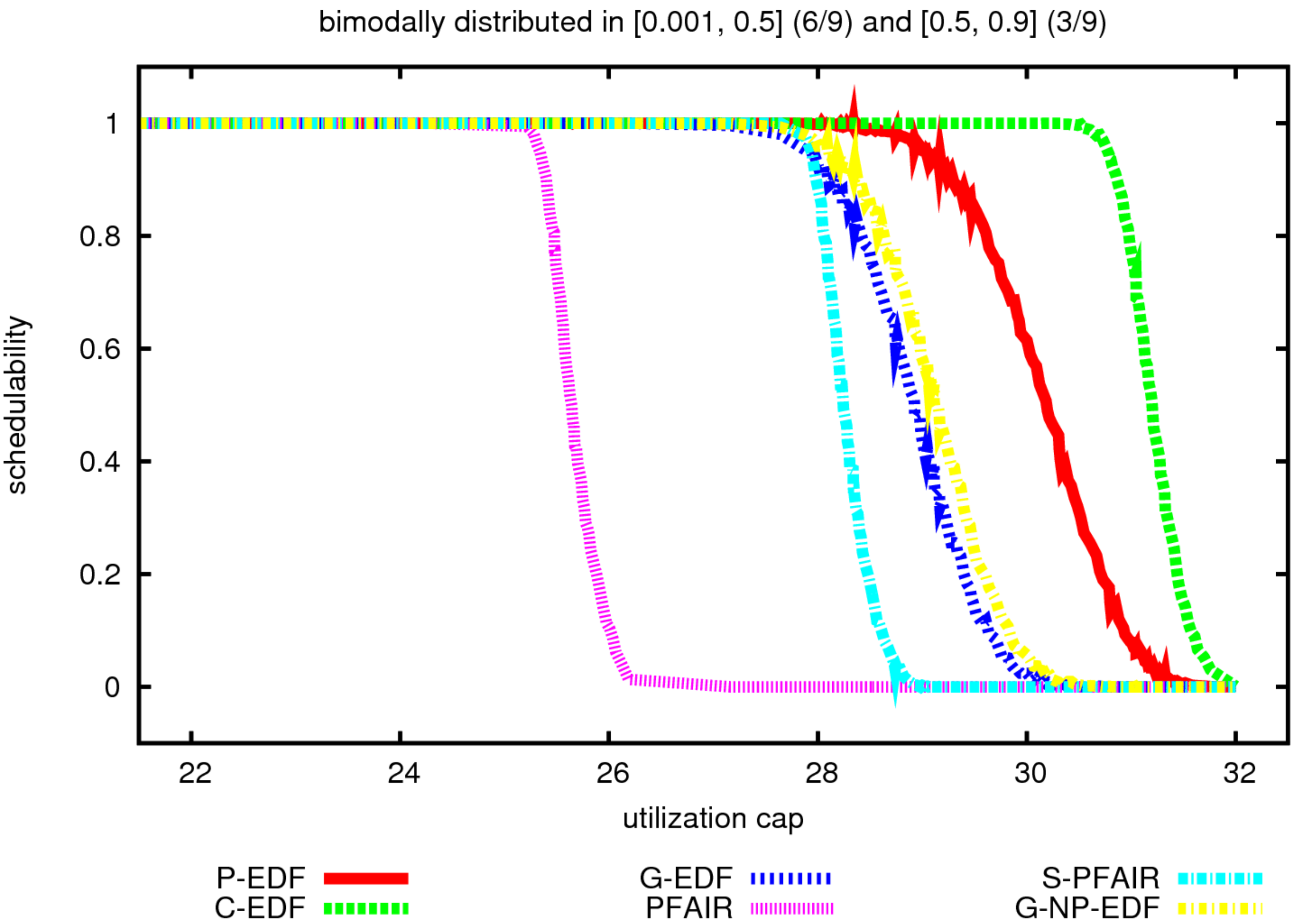


SRT, Bimodal Medium

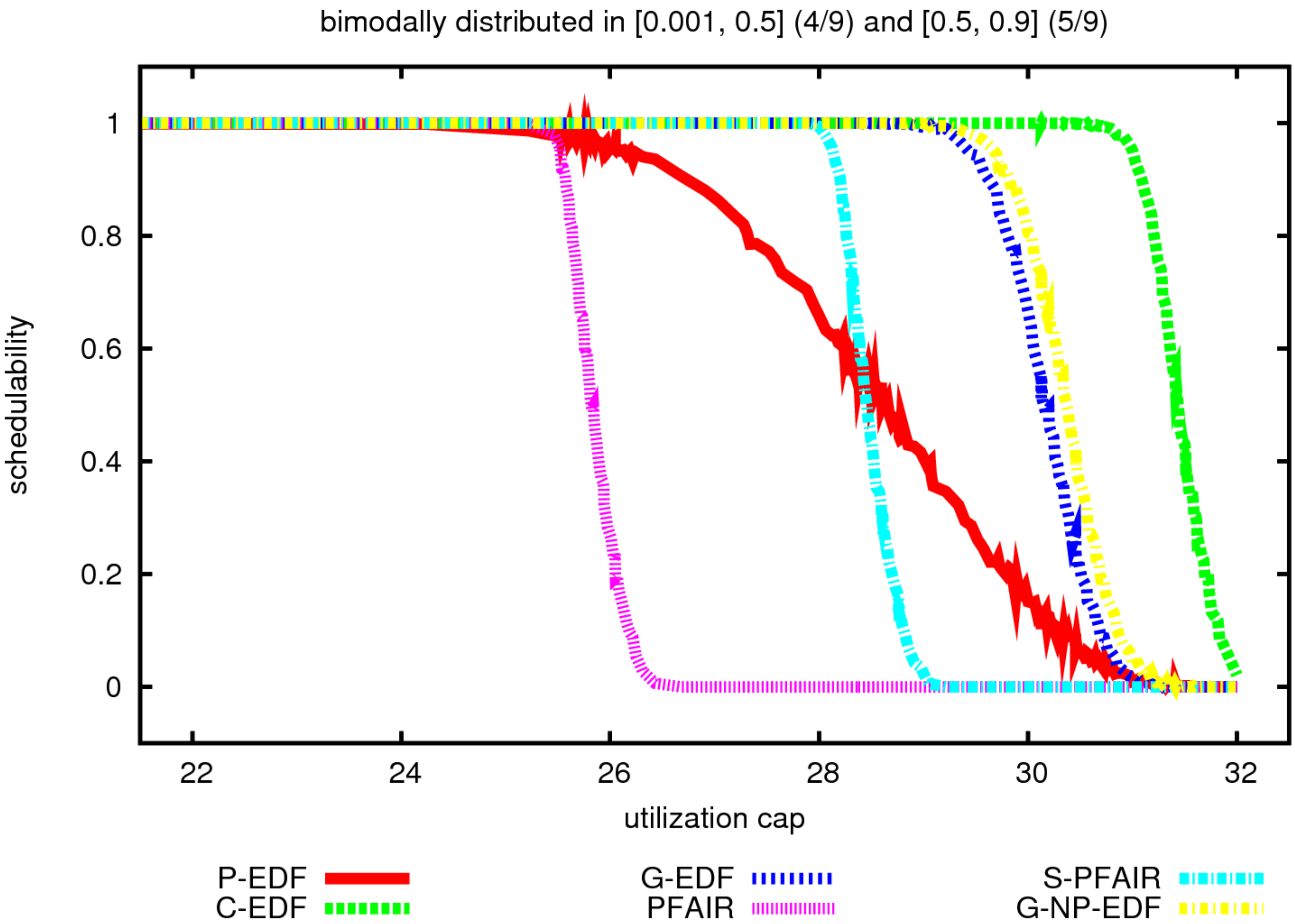


This and the next slide show that as the frequency of heavy tasks increases, PEDF degrades. CEDF isn’t affected by this increase much.

SRT, Bimodal Medium



SRT, Bimodal Heavy



Outline

- Background.
  - Real-time workload assumed.
  - Scheduling algorithms evaluated.
  - Some properties of these algorithms.
- Research questions addressed.
- Experimental results.
- Observations/speculation.
- Future work.

Observations/Speculation

- Global algorithms are really sensitive to how shared queues are implemented.
  - » Saw 100X performance improvement by switching from linked lists to binomial heaps.
  - » Still working on this…
  - » **Speculation:** Can reduce GEDF costs to close to PEDF costs for systems with ≤ 32 cores.
- Per algorithm, preempt. cost ≈ mig. cost.
  - » Due to having a shared cache.
  - » One catch: Migrations increase both costs.
- Quantum staggering is very effective.

Observations/Speculation (Cont'd)

- No one "best" algorithm.
- Intel has claimed they will produce an 80-core general-purpose chip. If they do…
  - » the cores will have to be simple ⇒ high execution costs ⇒ high utilizations ⇒ PEDF will suffer;
  - » "pure" global algorithms will not scale;
  - » some instantiation of CEDF (or maybe CS-PD^2) will hit the "sweet spot".

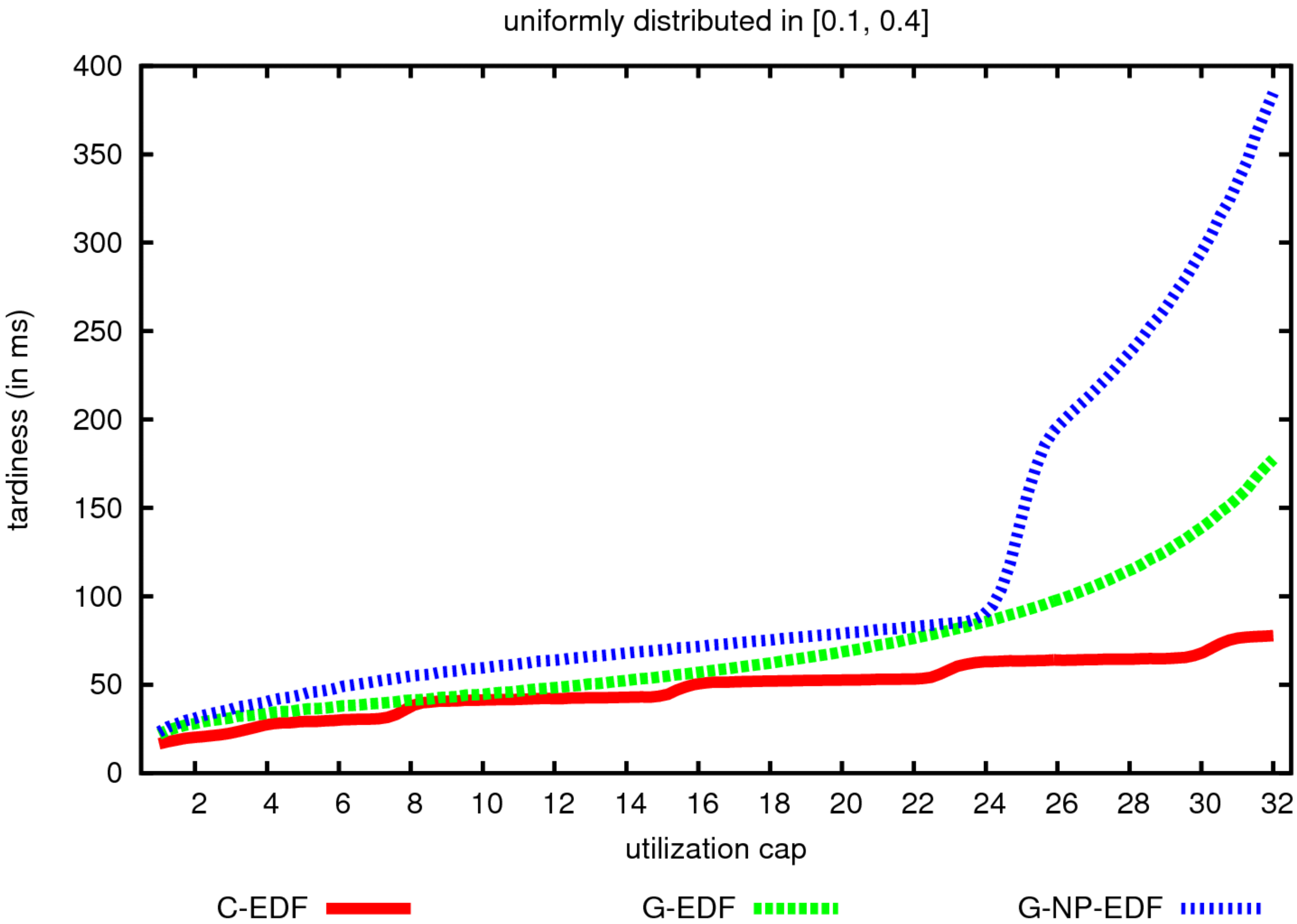
Future Work

- Thoroughly study “how to implement shared queues”.
- Repeat this study on Intel and embedded machines.
- Examine mixed HRT/SRT workloads.
- Factor in synchronization and dynamic behavior.
  - » In past work, PEDF was seen to be more negatively impacted by these things.

Thanks!

- Questions?

SRT Tardiness, Uniform Medium



Measuring Overheads

- Done using a UNC-produced tracer called Feather-Trace.
  - » http://www.cs.unc.edu/~bbb/feathertrace/
- Highest 1% of values were tossed.
  - » Eliminates "outliers" due to non-deterministic behavior in Linux, warm-up effects, etc.
- Used worst-case (average-case) values for HRT (SRT) schedulability.
- Used linear regression analysis to produce linear (in the task count) overhead expressions.

Obtaining Kernel Overheads

- Ran 90 (synthetic) task sets per scheduling algorithm for 30 sec.
- In total, over 600 million individual overheads were recorded (45 GB of data).

Kernel Overheads (in μs)

(N = no. of tasks)

Worst-Case

Alg	Tick	Schedule	Context SW	Release
PD^2	11.2 +.3N	32.7	3.1+.01N	---
$S\text{-}PD^2$	4.8+.3N	43.1	3.2+.003N	---
GEDF	3+.003N	55.2+.26N	29.2	45+.3N
CEDF	3.2	14.8+.01N	6.1	30.3
PEF	2.7+.002N	8.6+.01N	14.9+.04N	4.7+.009N

Average

Alg	Tick	Schedule	Context SW	Release
PD^2	4.3+.03N	4.7	2.6+.001N	---
$S\text{-}PD^2$	2.1+.02N	4.2	2.5+.001N	---
GEDF	2.1+.002N	11.8+.06N	7.6	5.8+.1N
CEDF	2.8	6.1+.01N	3.2	16.5
PEDF	2.1+.002N	2.7+.008N	4.7+.005N	4+.005N

Kernel Overheads (in μs)

(N = no. of tasks)

Worst-Case

Alg	Tick	Schedule	Context SW	Release
PD^2	11.2 +.3N	32.7	3.1+.01N	---
$S\text{-}PD^2$	4.8+.3N	43.1	3.2+.003N	---
GEDF	3+.003N	55.2+.26N	29.2	45+.3N
CEDF	3.2	14.8+.01N	6.1	30.3
PEF	2.7+.002N	8.6+.01N	14.9+.04N	4.7+.009N

Average

Alg	Tick	Schedule	Context SW	Release
PD^2	4.3+.03N	4.7	2.6+.001N	---
$S\text{-}PD^2$	2.1+.02N	4.2	2.5+.001N	---
GEDF	2.1+.002N	11.8+.06N	7.6	5.8+.1N
CEDF	2.8	6.1+.01N	3.2	16.5
PEDF	2.1+.002N	2.7+.008N	4.7+.005N	4+.005N

Obtaining Preemption/Migration Overheads

- Ran 90 (synthetic) task sets per scheduling algorithm for 60 sec.
- Each task has a 64K working set (WS) that it accesses repeatedly with a 75/25 read/write ratio.
- Recorded time to access WS after preemption/migration minus “cache-warm access”.
- In total, over 105 million individual preemption/ migration overheads were recorded (15 GB of data).

Preemption/Migration Overheads (in μs)

(N = no. of tasks)

Worst-Case

Alg	Overall	Preemption	Intra-Cluster Mig	Inter-Cluster Mig
PD^2	681.1	649.4	654.2	681.1
S-PD^2	104.1	103.4	103.4	104.1
GEDF	375.4	375.4	326.8	321.1
CEDF	171.6	171.6	167.3	---
PEDF	139.1	139.1	---	---

Average

Alg	Overall	Preemption	Intra-Cluster Mig	Inter-Cluster Mig
PD^2	172	131.4	141.8	187.6
S-PD^2	89.3	86.2	87.8	90.2
GEDF	73	95.1	73.5	72.6
CEDF	67	78.5	64.8	---
PEDF	72.3	72.3	---	---

Preemption/Migration Overheads (in μs)

(N = no. of tasks)

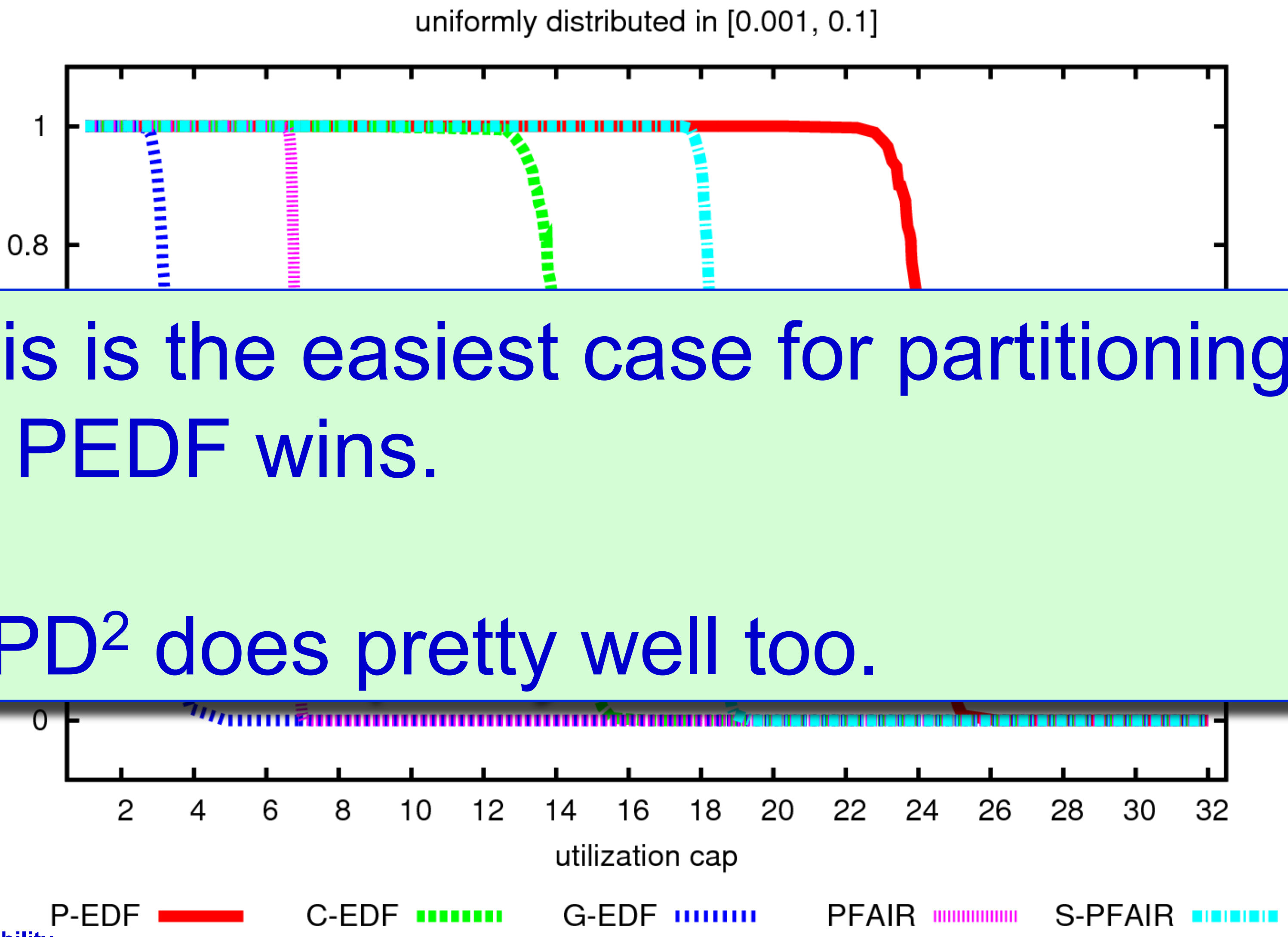
Worst-Case

Alg	Overall	Preemption	Intra-Cluster Mig	Inter-Cluster Mig
PD^2	681.1	649.4	654.2	681.1
$S\text{-}PD^2$	104.1	103.4	103.4	104.1
GEDF	375.4	375.4	326.8	321.1
CEDF	171.6	171.6	167.3	---
PEDF	139.1	139.1	---	---

Average

Alg	Overall	Preemption	Intra-Cluster Mig	Inter-Cluster Mig
PD^2	172	131.4	141.8	187.6
$S\text{-}PD^2$	89.3	86.2	87.8	90.2
GEDF	73	95.1	73.5	72.6
CEDF	67	78.5	64.8	---
PEDF	72.3	72.3	---	---

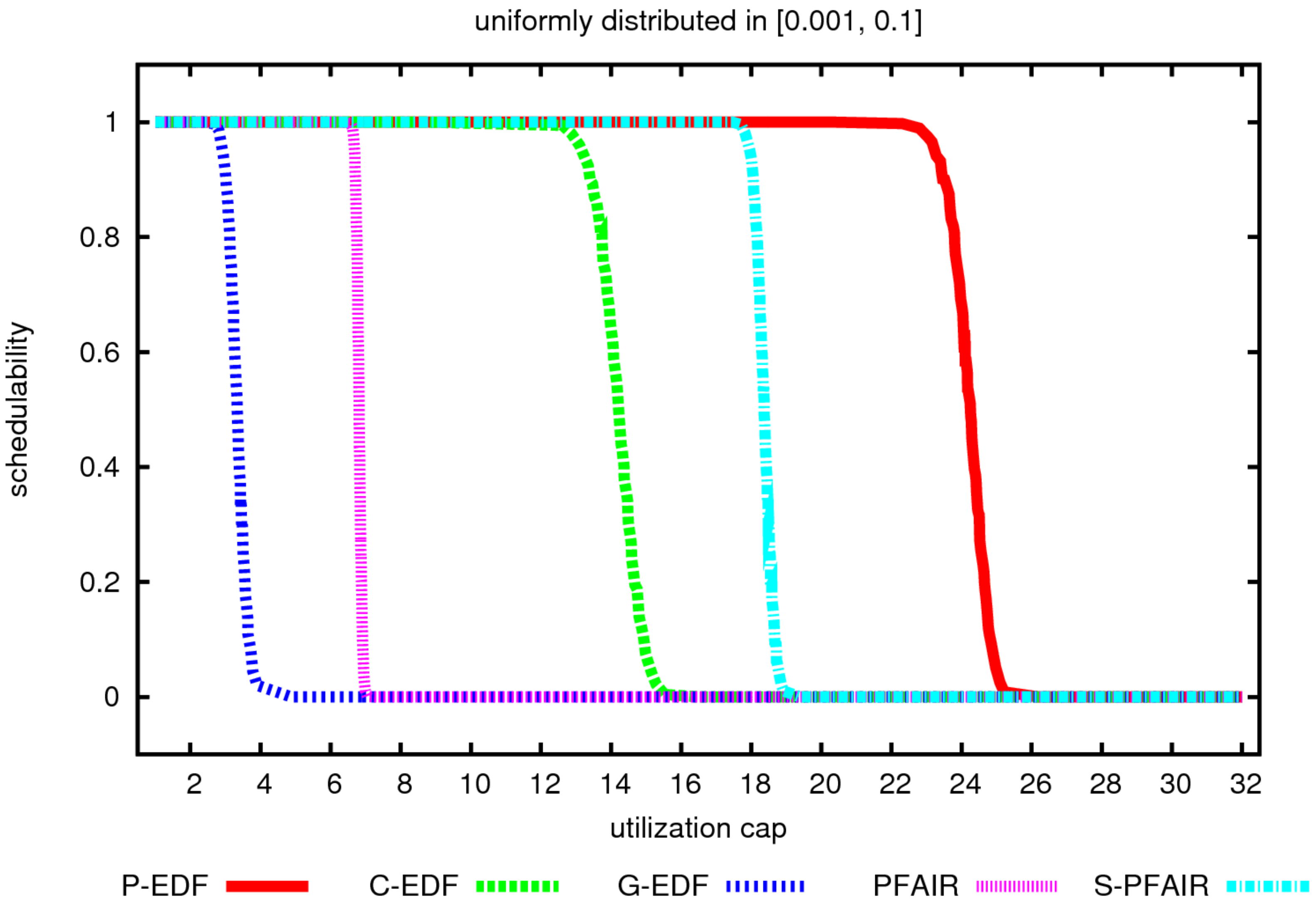
HRT, Uniform Light



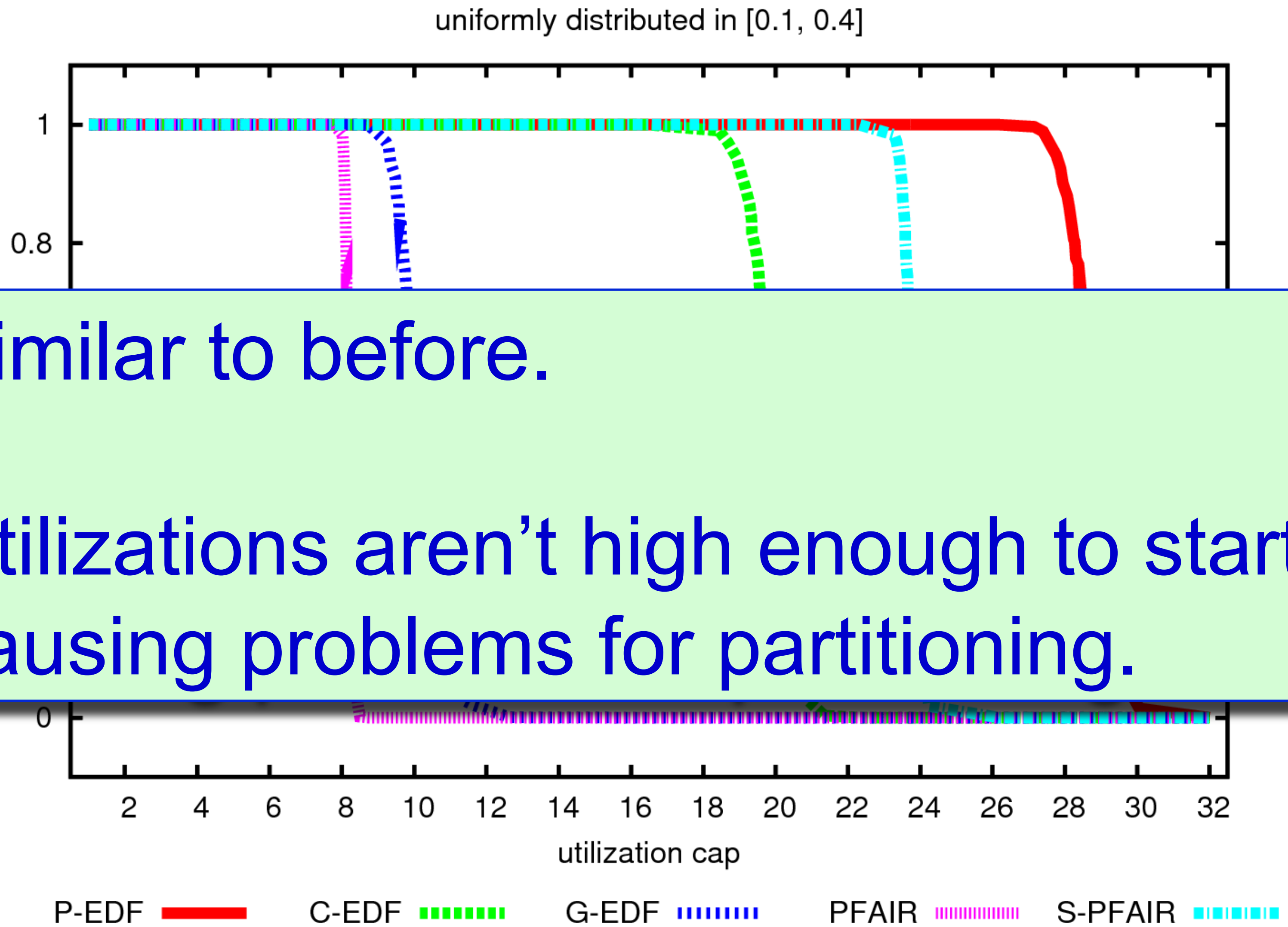
This is the easiest case for partitioning, so PEDF wins.

S-PD^2 does pretty well too.

HRT, Uniform Light



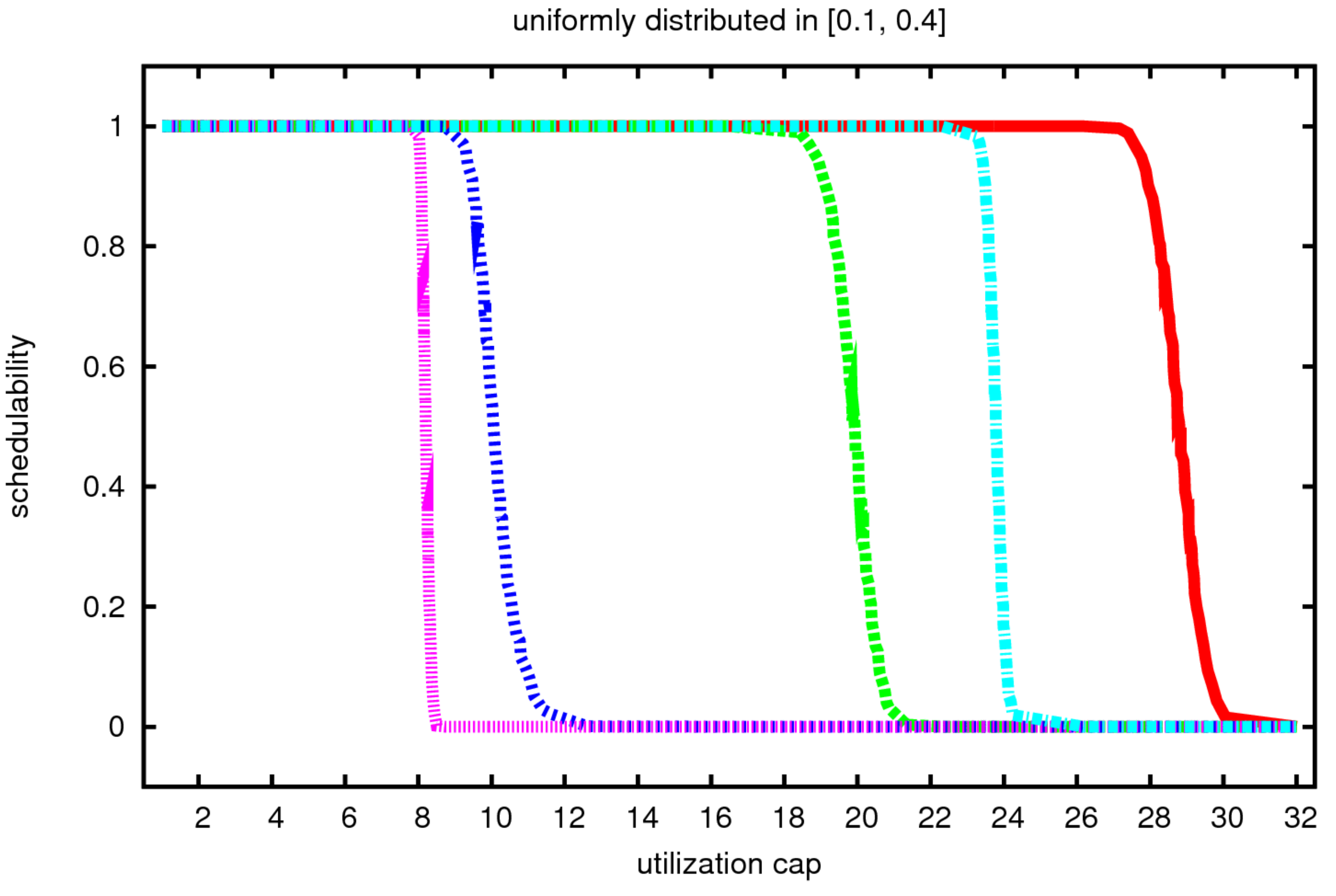
HRT, Uniform Medium



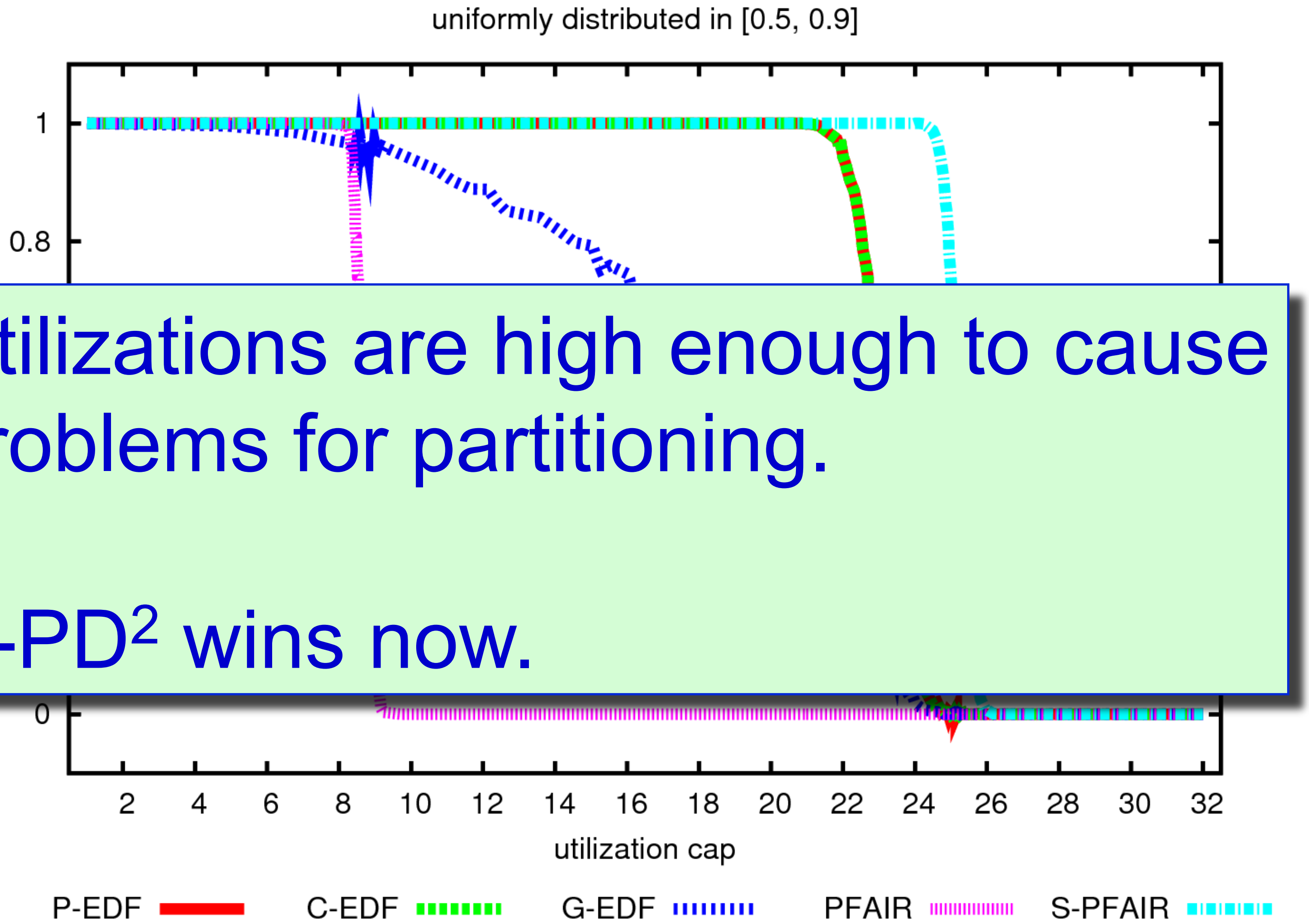
Similar to before.

Utilizations aren’t high enough to start causing problems for partitioning.

HRT, Uniform Medium



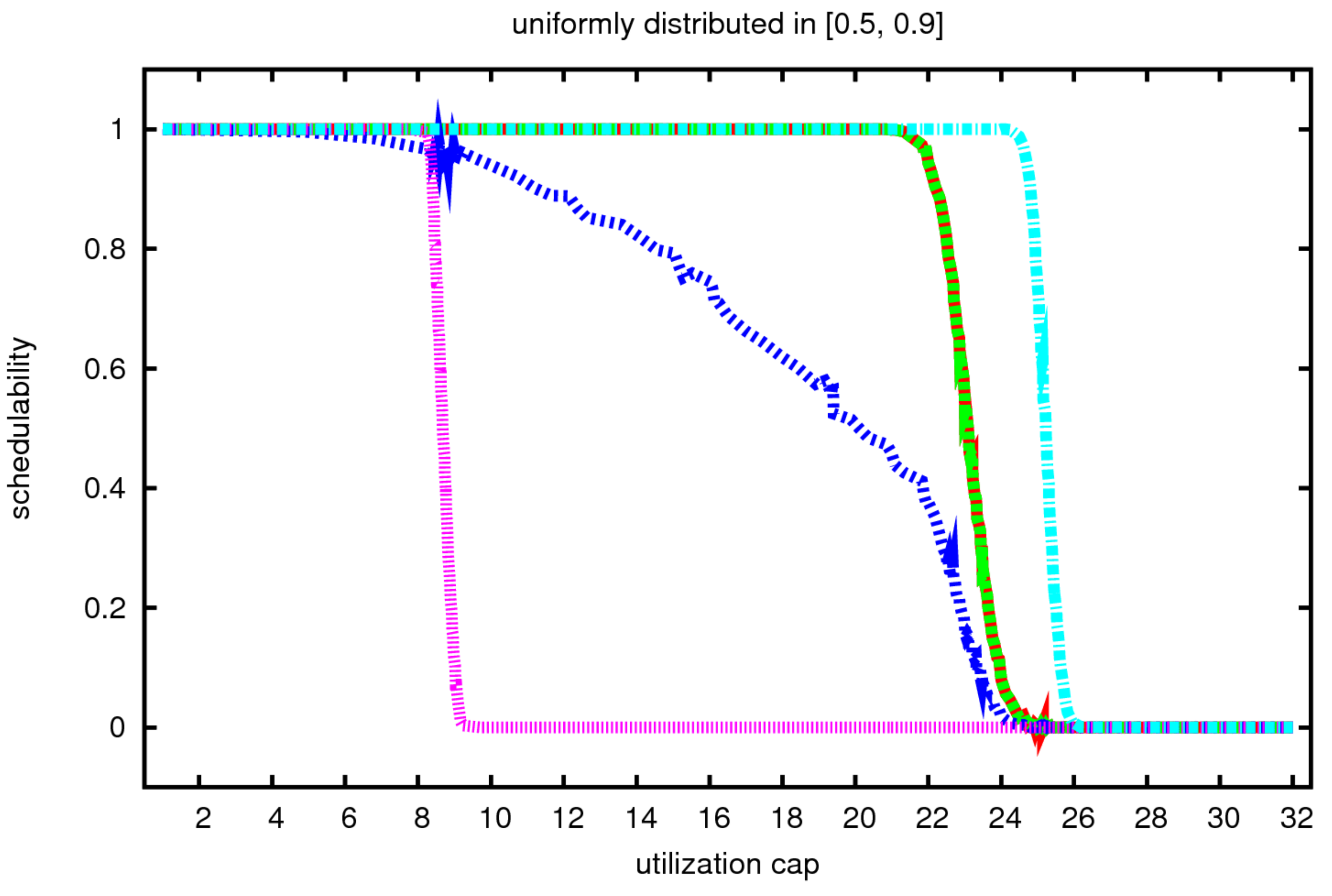
HRT, Uniform Heavy



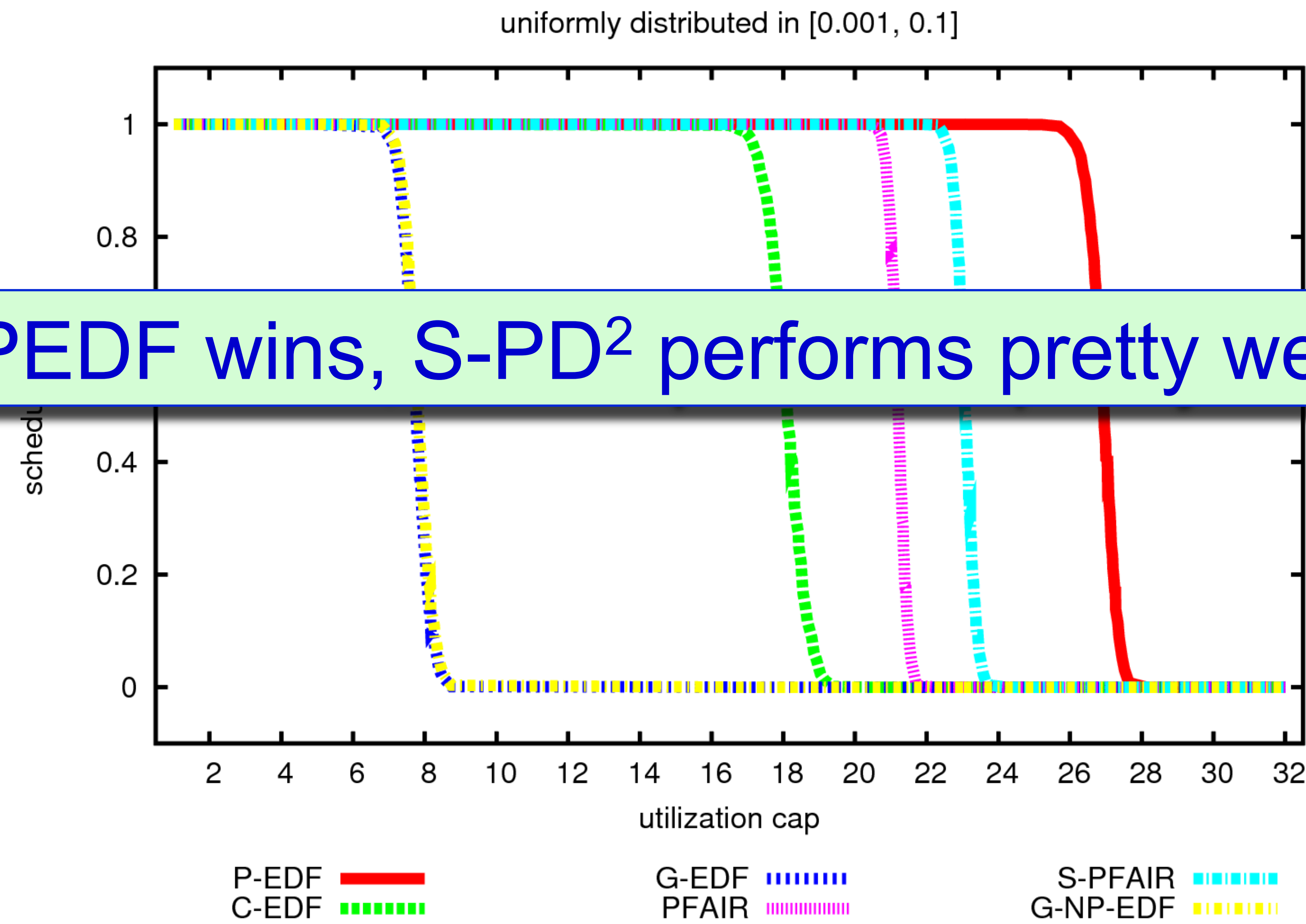
Utilizations are high enough to cause problems for partitioning.

S-PD^2 wins now.

HRT, Uniform Heavy

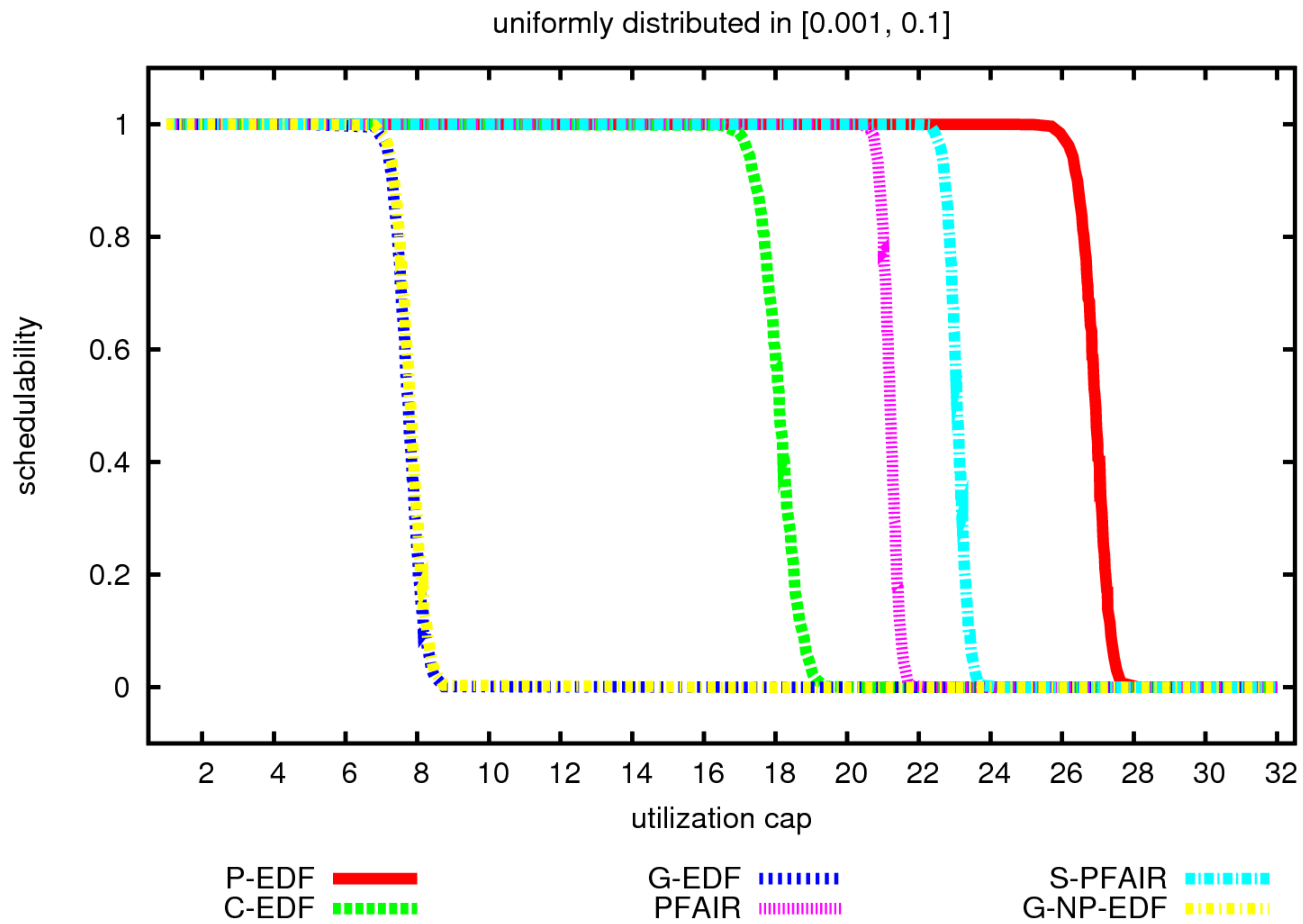


SRT, Uniform Light

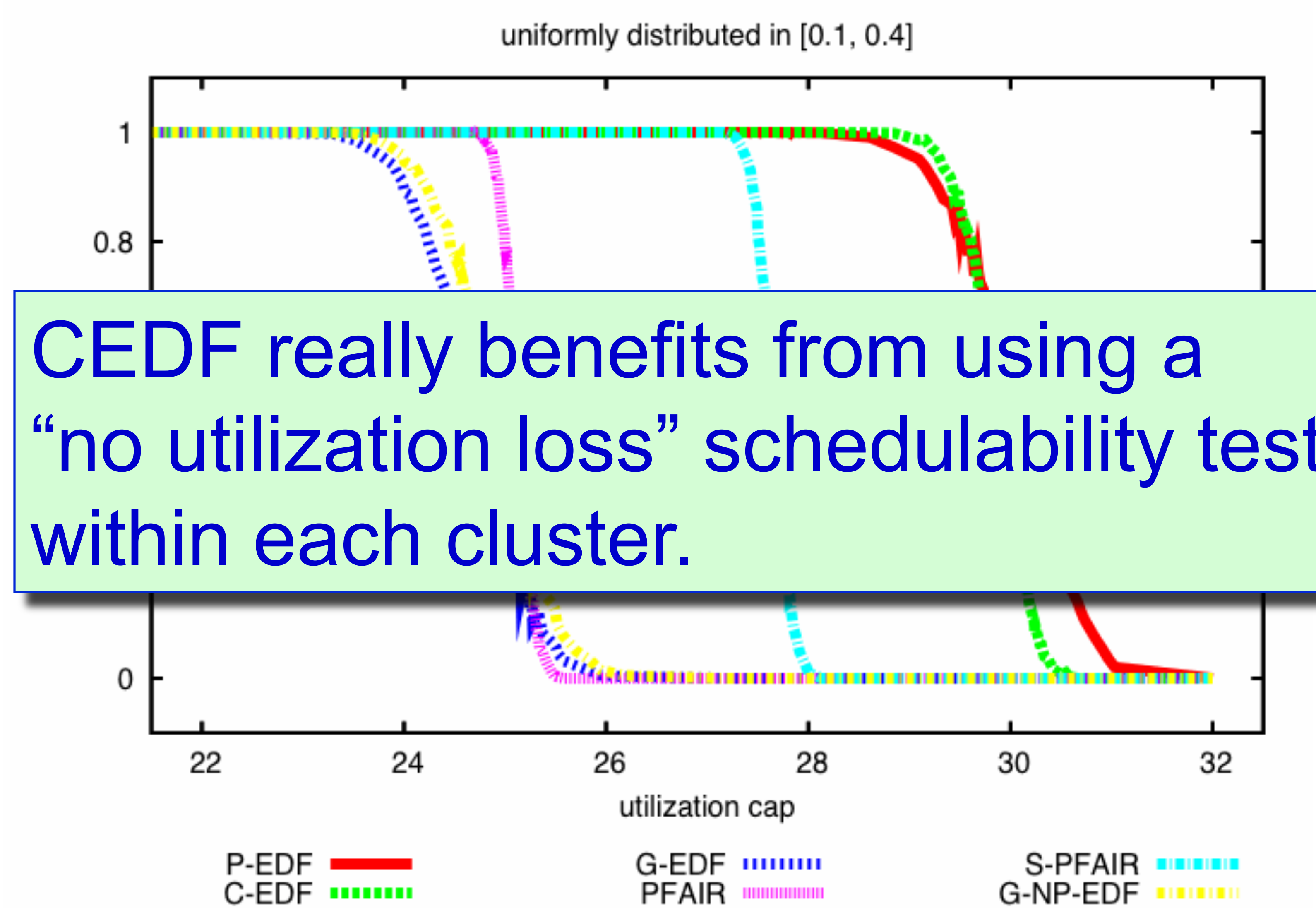


PEDF wins, S-PD^2 performs pretty well.

SRT, Uniform Light

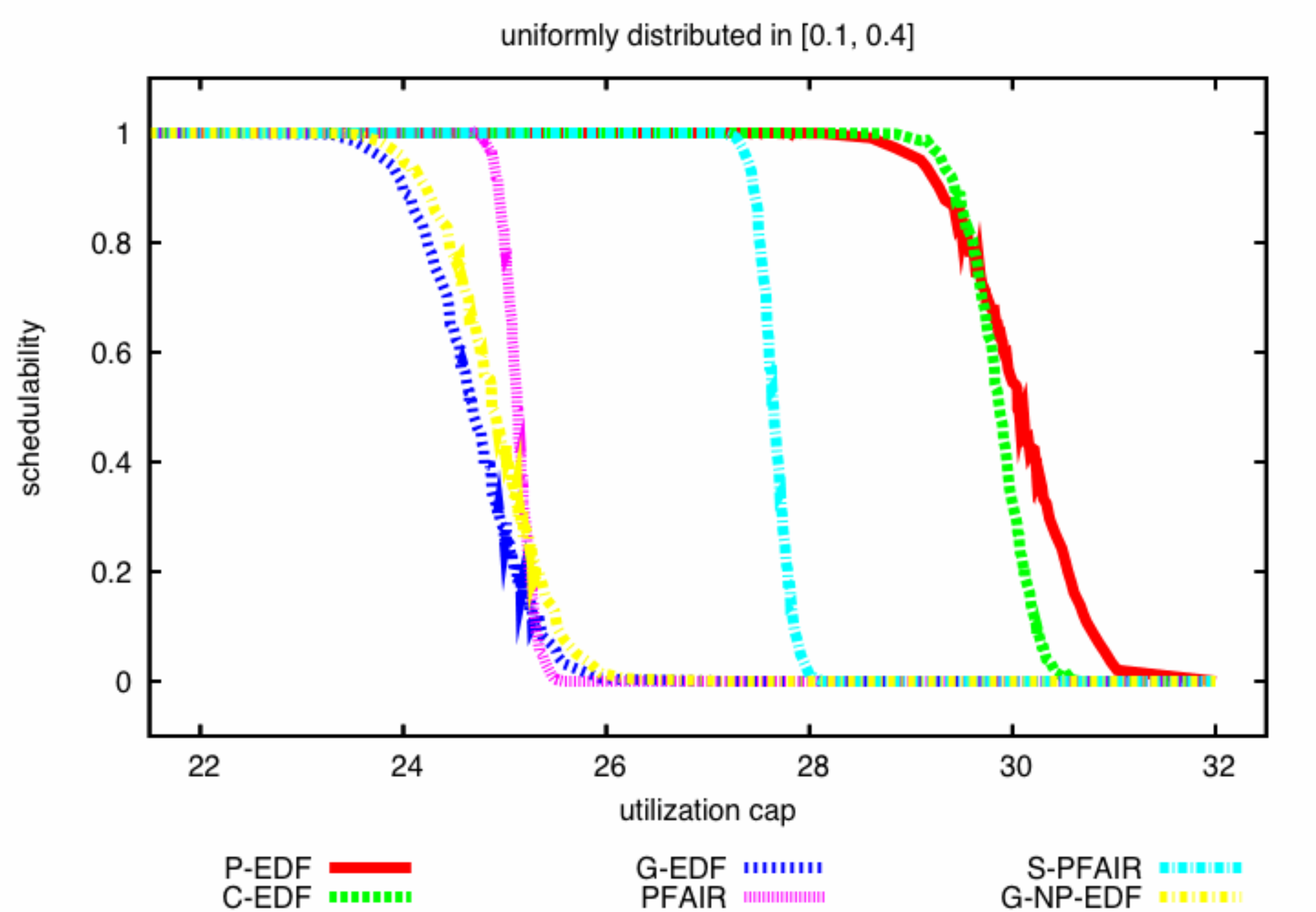


SRT, Uniform Medium

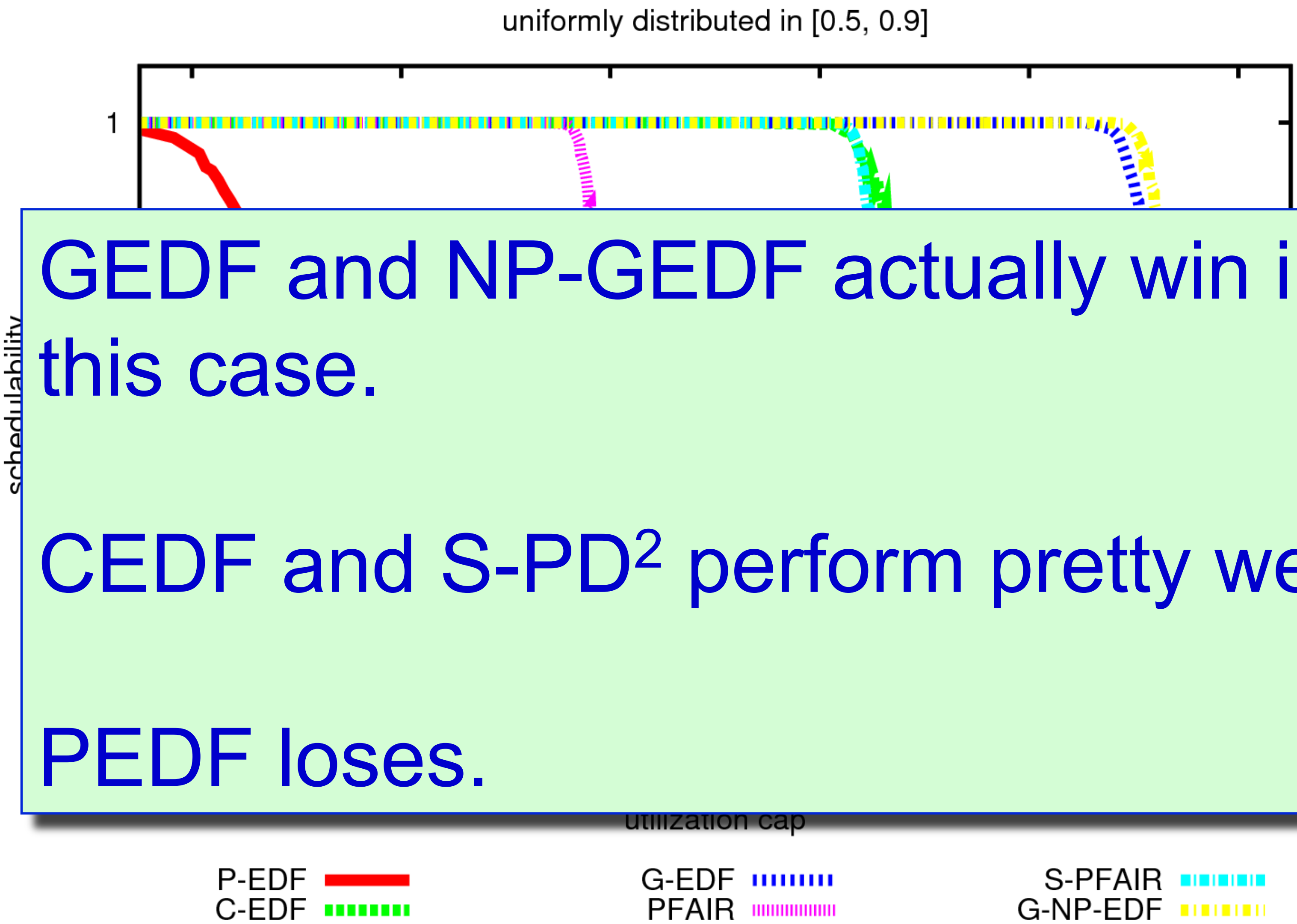


CEDF really benefits from using a “no utilization loss” schedulability test within each cluster.

SRT, Uniform Medium



SRT, Uniform Heavy

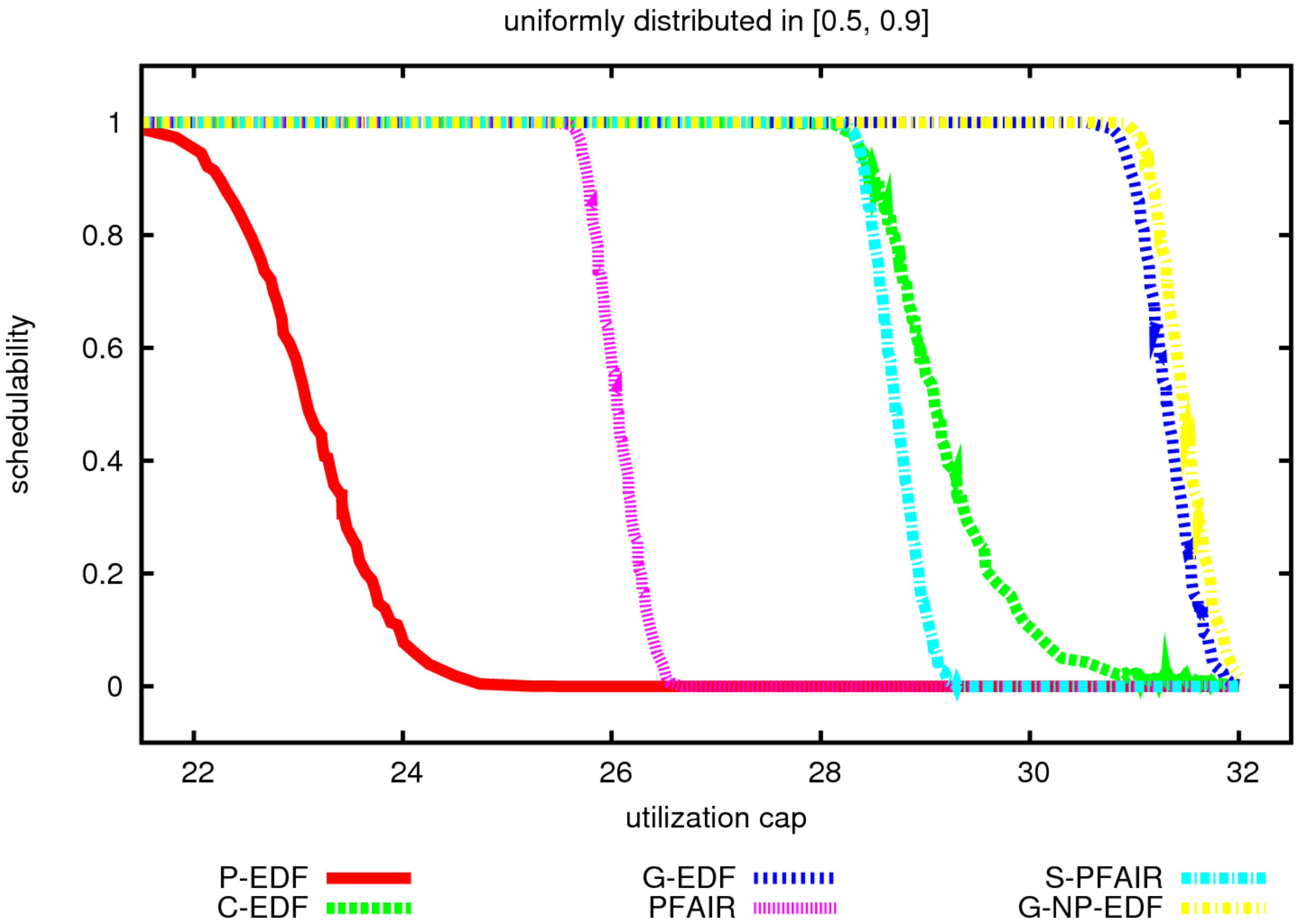


GEDF and NP-GEDF actually win in this case.

CEDF and S-PD^2 perform pretty well.

PEDF loses.

SRT, Uniform Heavy



On the Implementation of Global Real-Time Schedulers

Simon Fraser University
April 15, 2010

Sathish Gopalakrishnan
The University of British Columbia

Work supported by IBM, SUN, and Intel Corps., NSF grants CNS 0834270, CNS 0834132, and CNS 0615197, and ARO grant W911NF-06-1-0425.

UNC's Implementation Studies (I)

Calandrino et al. (2006)

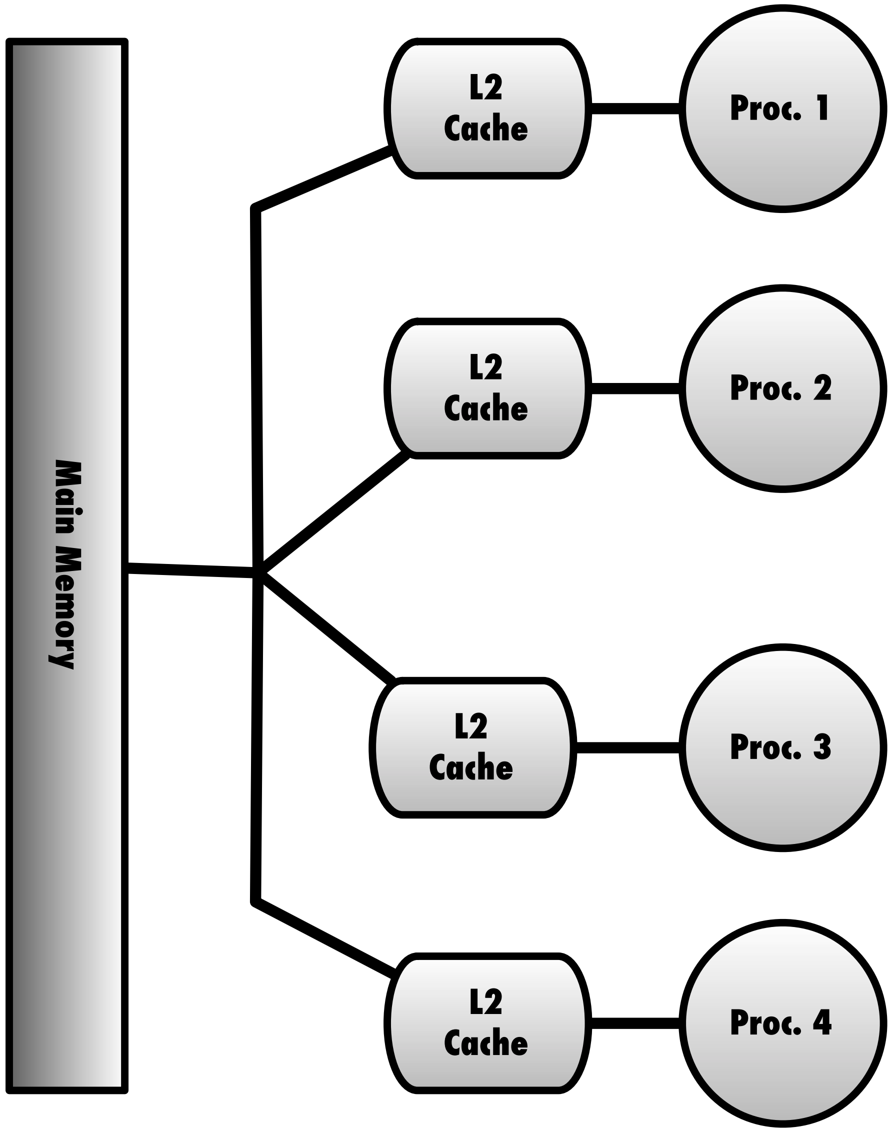
- ➡ Are commonly-studied RT schedulers **implementable**?
- ➡ In Linux on common hardware platforms?

Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

UNC's Implementation Studies (I)

Calandrino et al. (2006)

- ➡ Are commonly-studied RT schedulers **implementable**?
- ➡ In Linux on common hardware platforms?



Intel 4x 2.7 GHz Xeon SMP

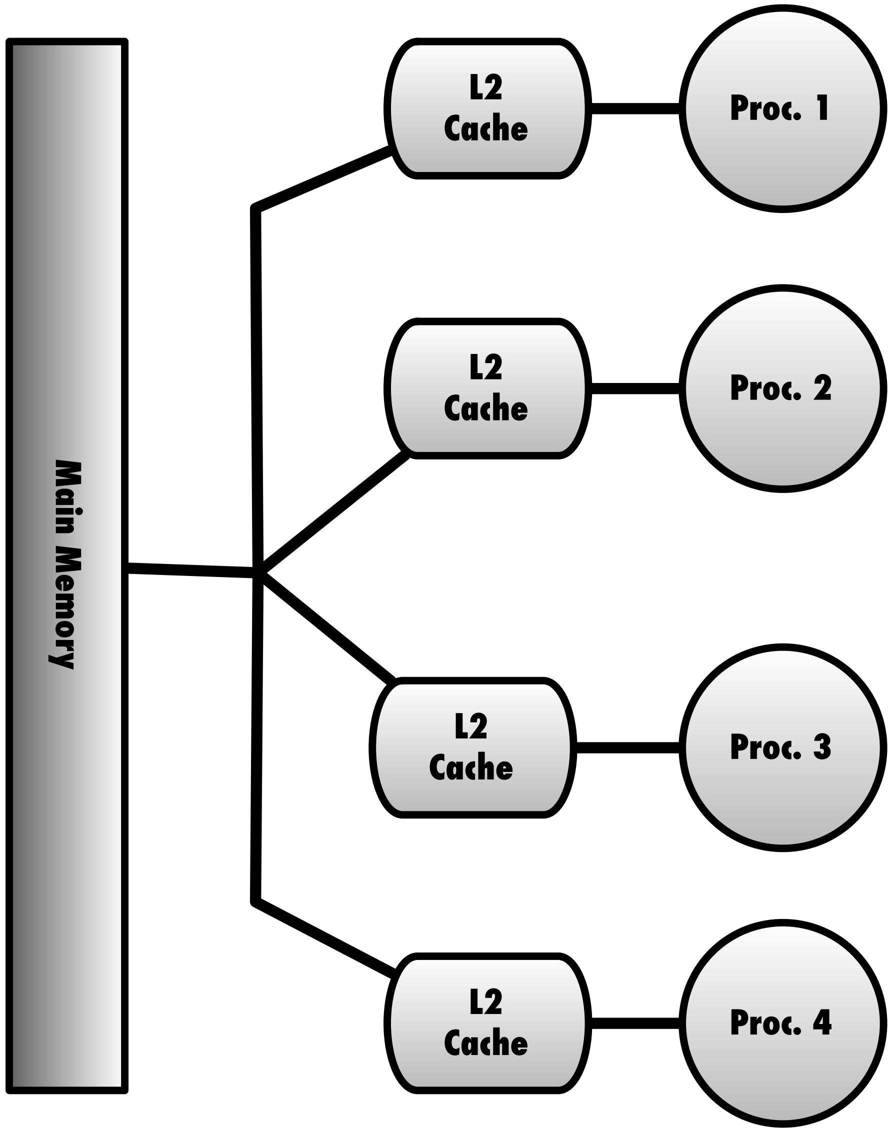
(few, fast processors; private caches)

Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

UNC's Implementation Studies (I)

Calandrino et al. (2006)

- ➡ Are commonly-studied RT schedulers **implementable**?
- ➡ In Linux on common hardware platforms?



partitioned EDF
P-EDF

2 x global EDF
G-NP-EDF
G-EDF

2 x PFAIR
PD²
S-PD²

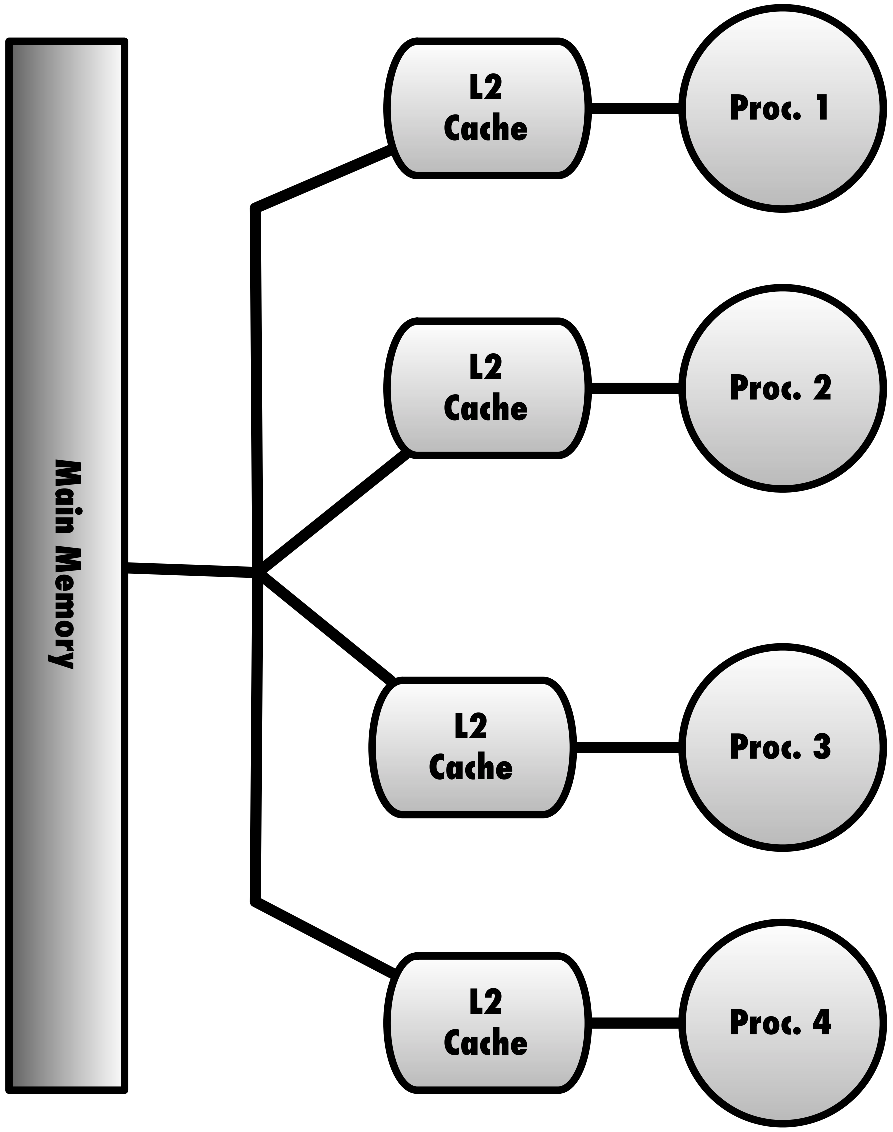
Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

UNC's Implementation Studies (I)

> ***"for each tested scheme, scenarios exist in which it is a viable choice"***

Calandrino

- ➡ Are comm
- ➡ In Linux on common hardware platforms?



P-EDF

G-NP-EDF

G-EDF

PD^2

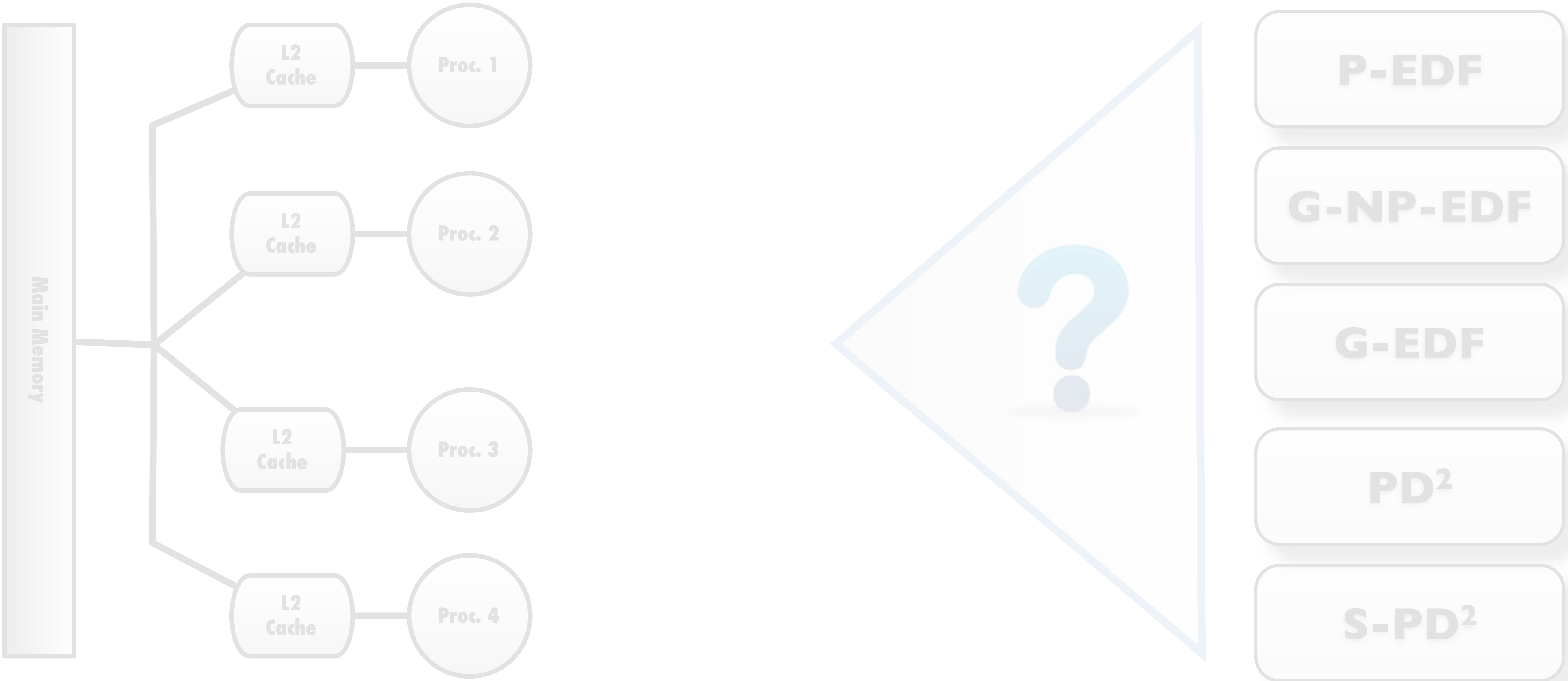
S-PD^2

Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

UNC's Implementation Studies (II)

Brandenburg et al. (2008)

➡ What if there are **many slow processors**?

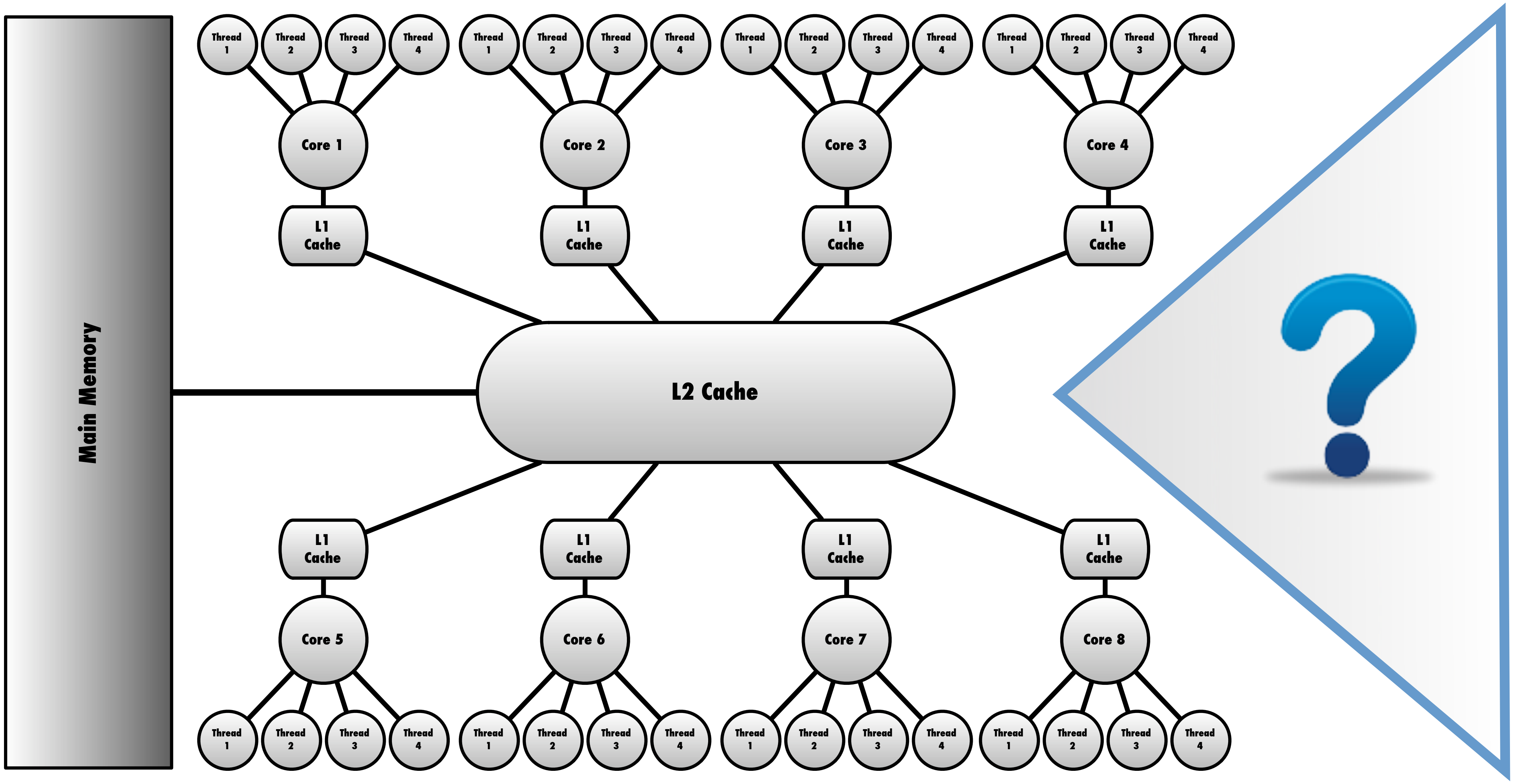


Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

UNC’s Implementation Studies (II)

Brandenburg et al. (2008)

- ➡ What if there are **many slow processors**?
- ➡ Explored **scalability** of RT schedulers on a Sun Niagara.



- P-EDF
- G-NP-EDF
- G-EDF
- PD^2
- S-PD^2

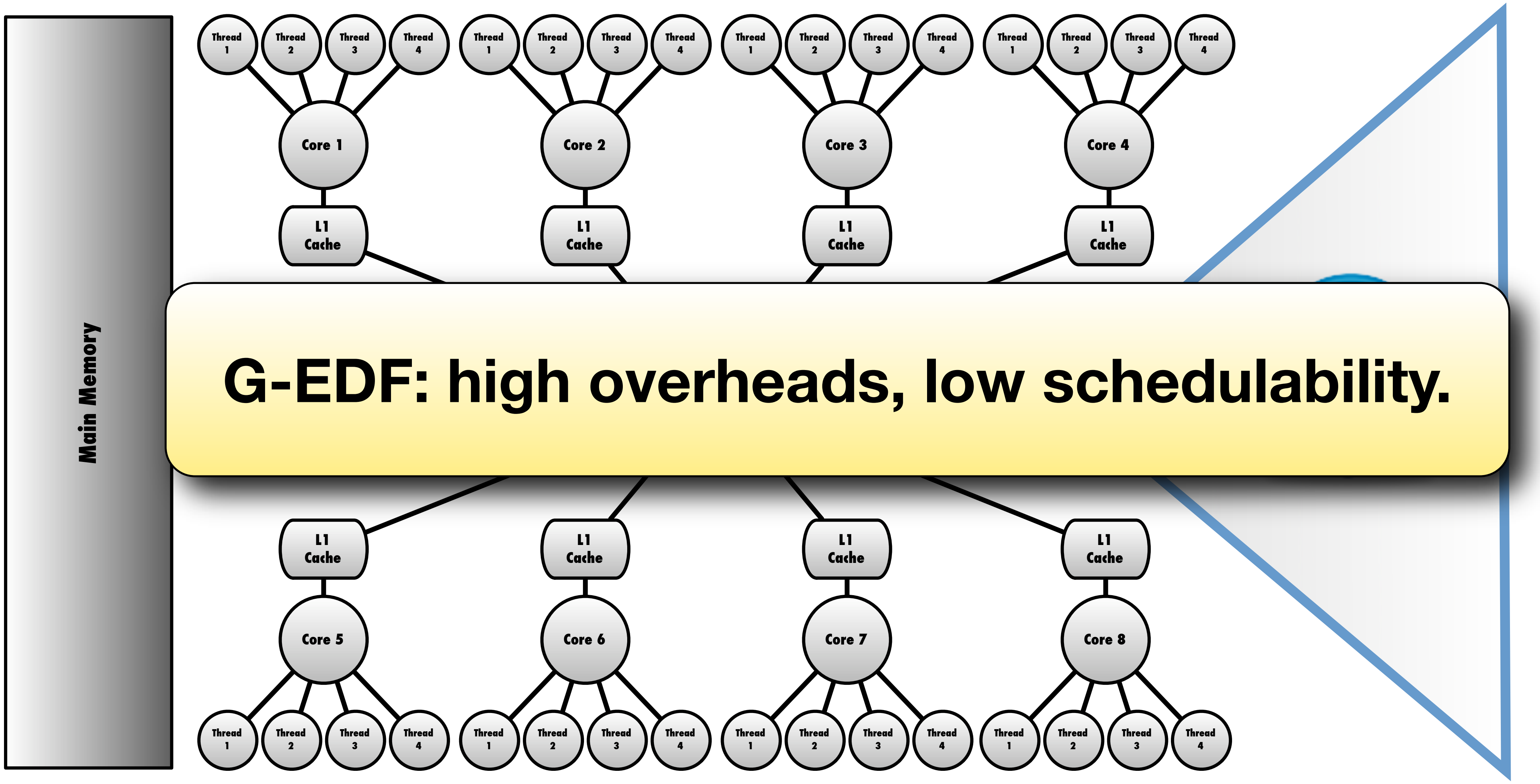
Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

UNC’s Implementation Studies (II)

Brandenburg et al. (2008)

➡ What if there are **many slow processors**?

➡ Explored **scalability** of RT schedulers on a Sun Niagara.



G-EDF: high overheads, low schedulability.

P-EDF

G-NP-EDF

G-EDF

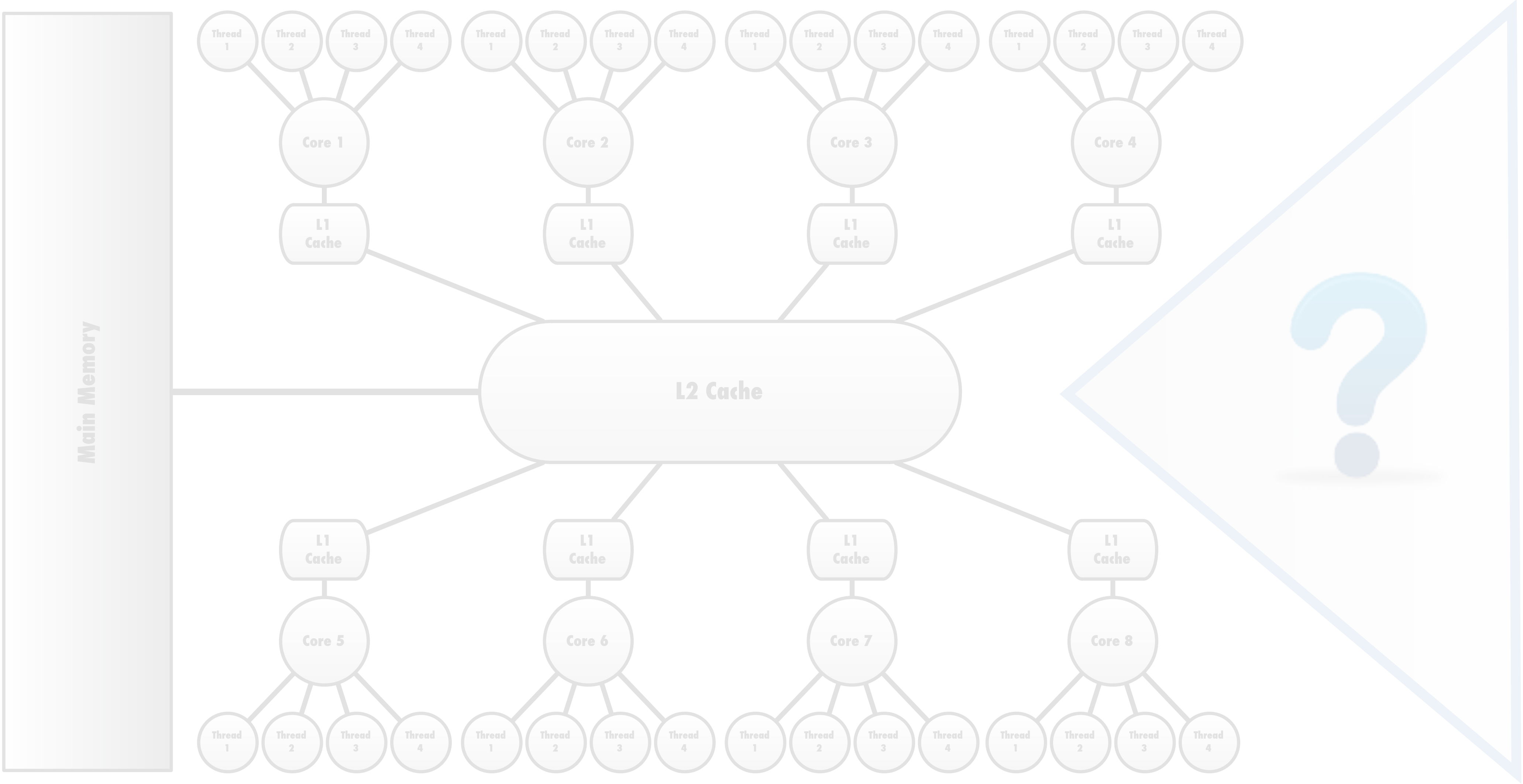
PD^2

S-PD^2

Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

Today’s discussion

How to implement global schedulers?



P-EDF

G-NP-EDF

G-EDF

PD^2

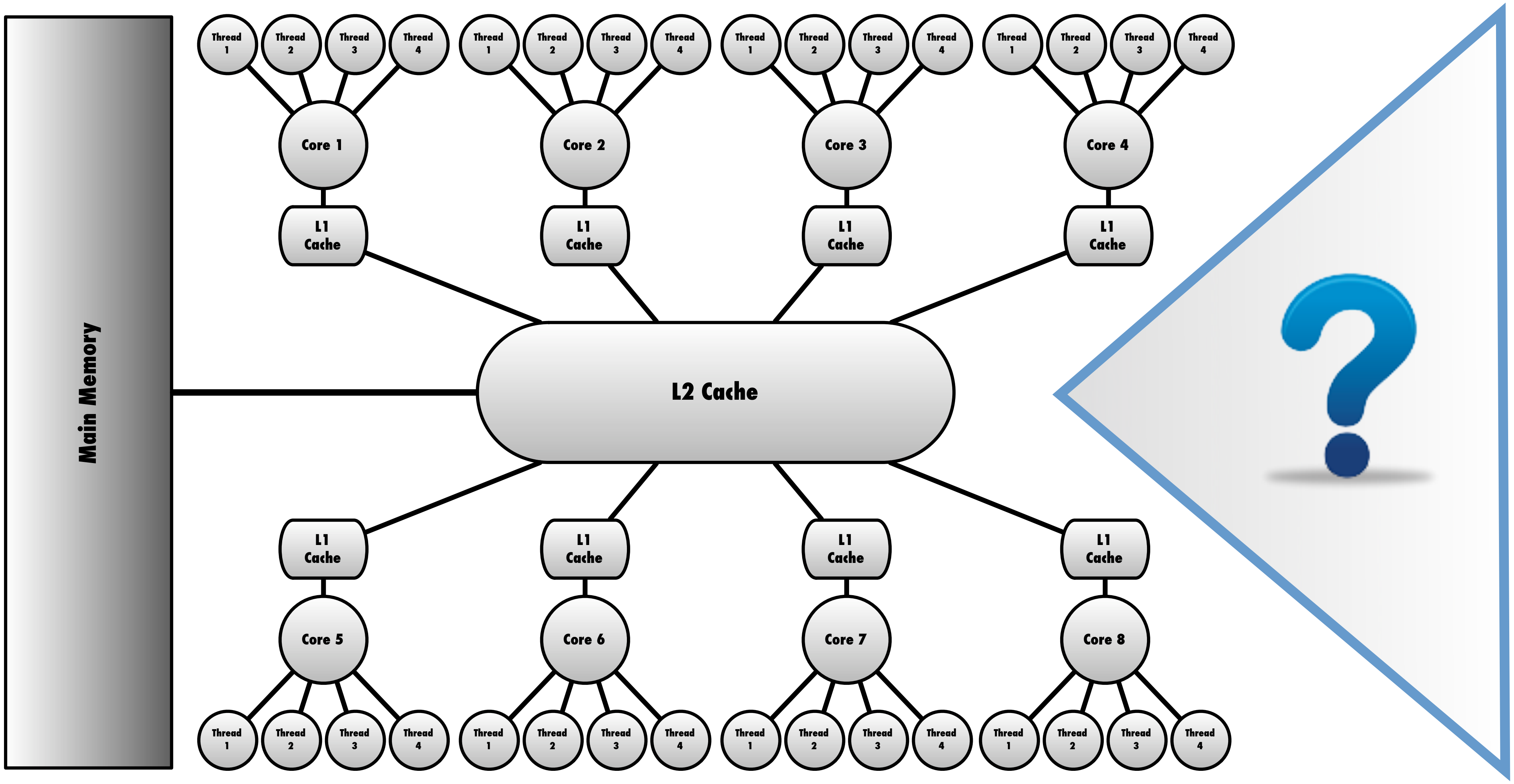
S-PD^2

Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

Today’s discussion

How to implement global schedulers?

➡ Explore how **implementation tradeoffs** affect **schedulability**.



Instead of considering **one** implementation of several **different** scheduling algorithms…

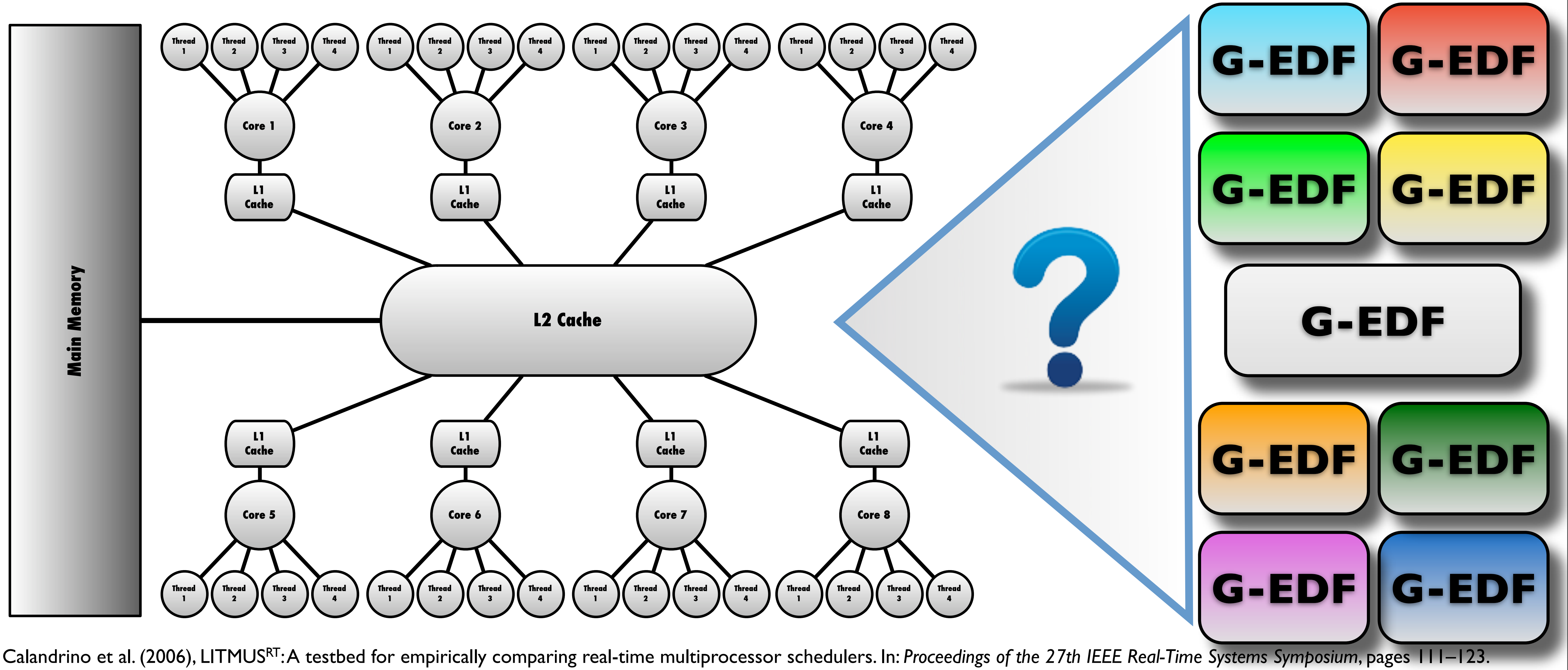
Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.

Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

Today's discussion

How to implement global schedulers?

- ➡ Explore how **implementation tradeoffs** affect **schedulability**.
- ➡ Case study: **nine G-EDF variants** on a Sun Niagara.



Calandrino et al. (2006), LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123.
Brandenburg et al. (2008), On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

Design Choices

Design Choices

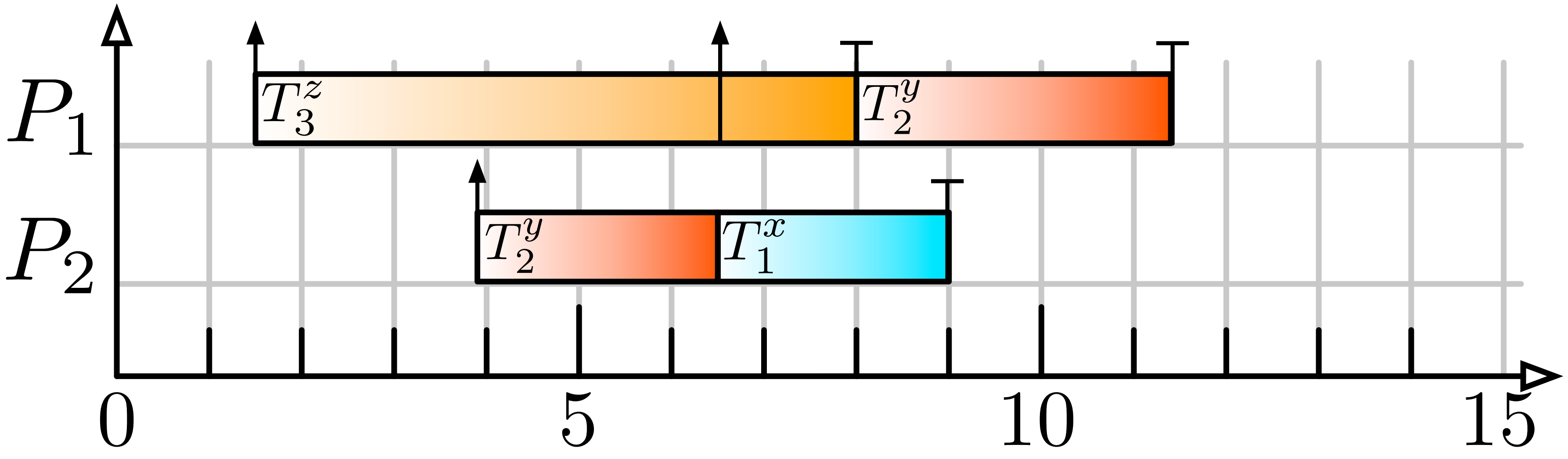
- ➡ When to schedule.
- ➡ Quantum alignment.
- ➡ How to handle interrupts.
- ➡ How to queue pending jobs.
- ➡ How to manage future releases.
- ➡ How to avoid unnecessary preemptions.

Scheduler Invocation

Scheduler Invocation

Event-Driven

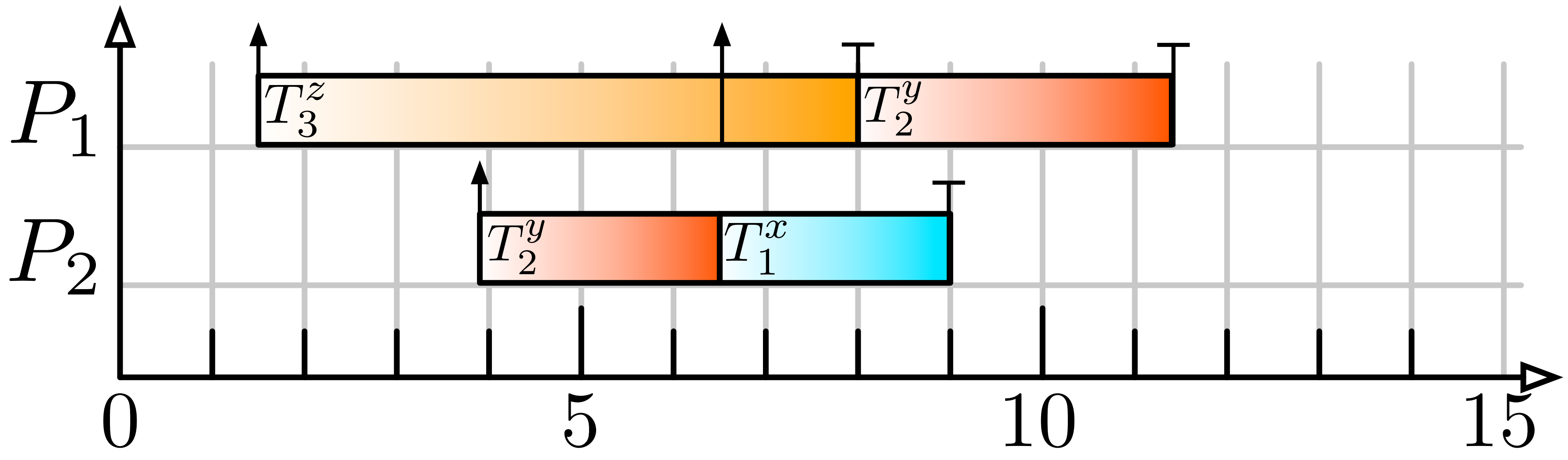
- ➡ on job release
- ➡ on job completion
- ➡ preemptions occur immediately



Scheduler Invocation

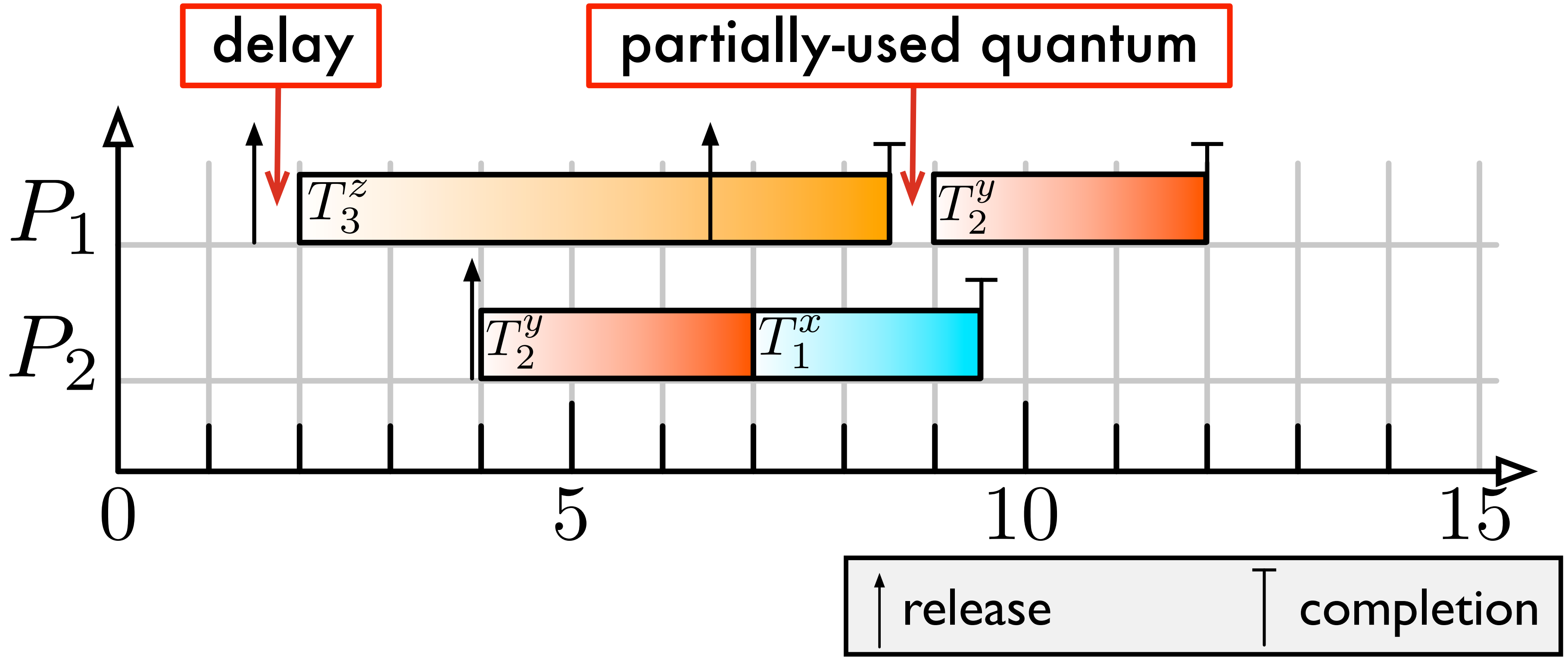
Event-Driven

- on job release
- on job completion
- preemptions occur immediately



Quantum-Driven

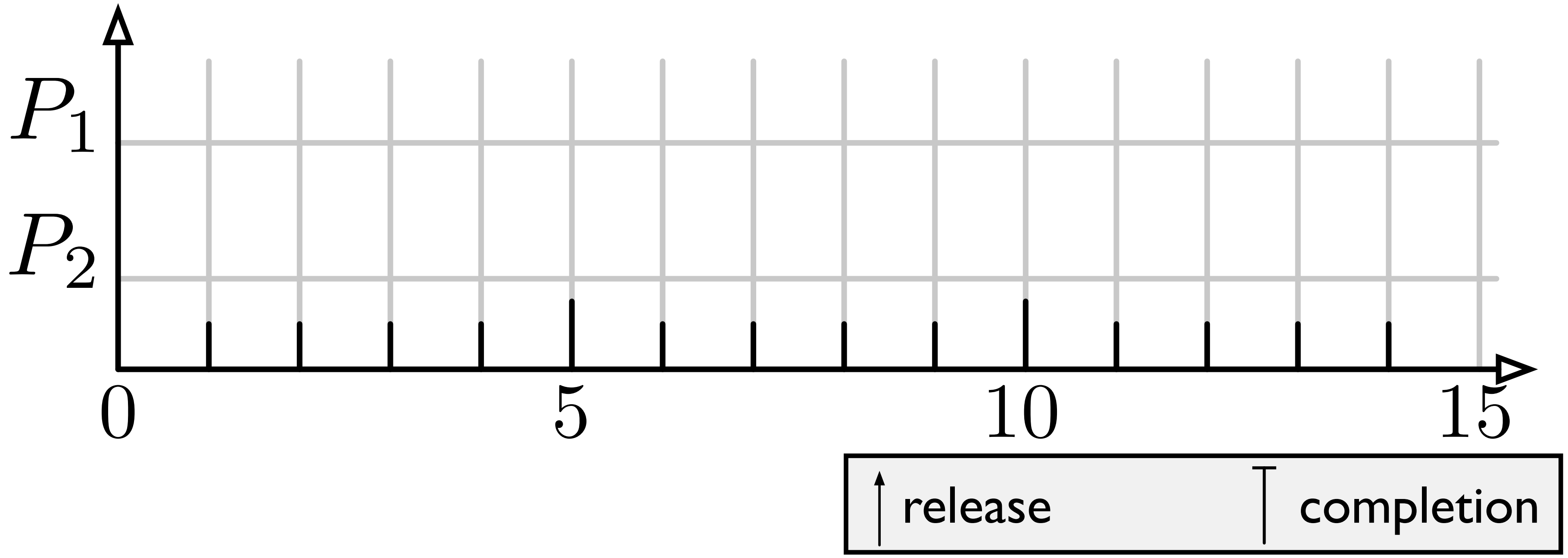
- on every timer tick
- easier to implement
- on release a job is just enqueued; scheduler is invoked at next tick



Quantum Alignment

Aligned

- ➡ Tick **synchronized** across processors.
- ➡ **Contention** at quantum boundary!



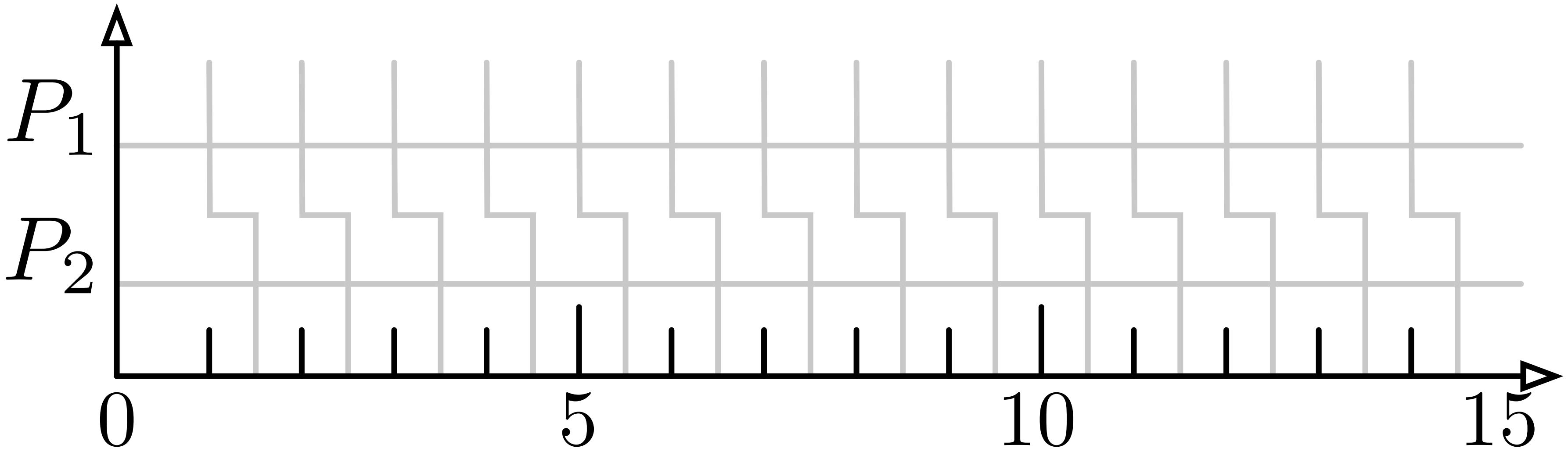
Quantum Alignment

Staggered

➡ Ticks spread out across quantum.

➡ **Reduced** bus and lock contention.

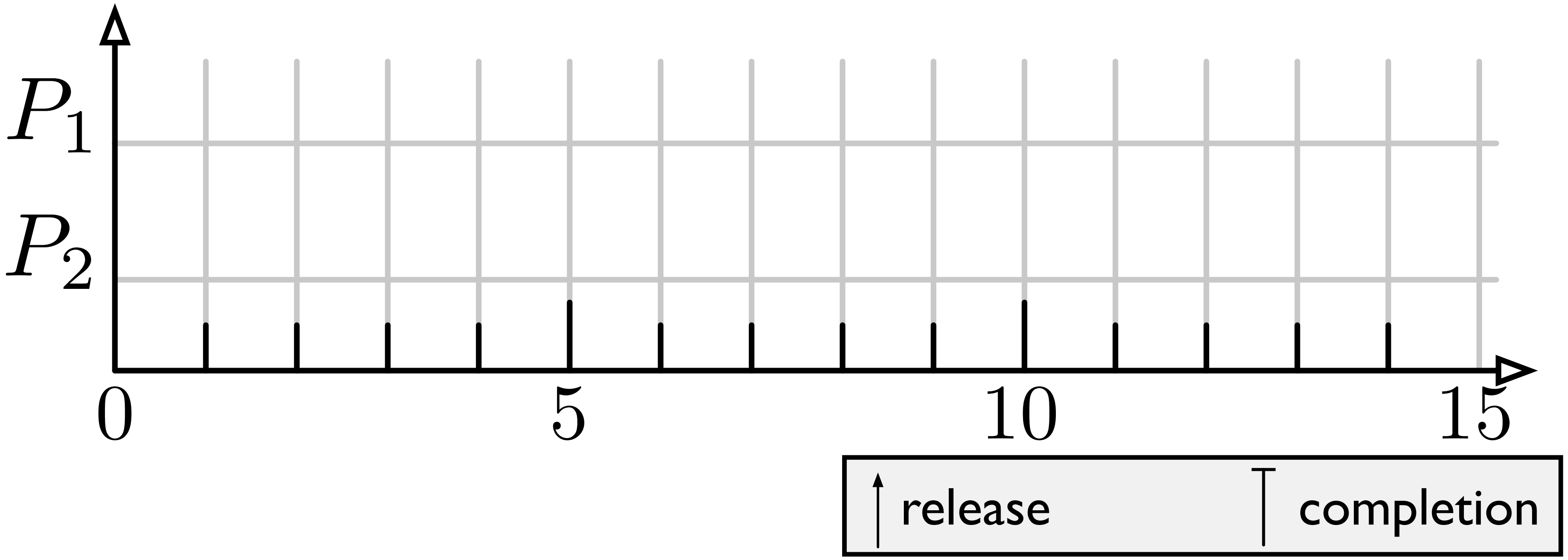
➡ Additional **latency**.



Aligned

➡ Tick **synchronized** across processors.

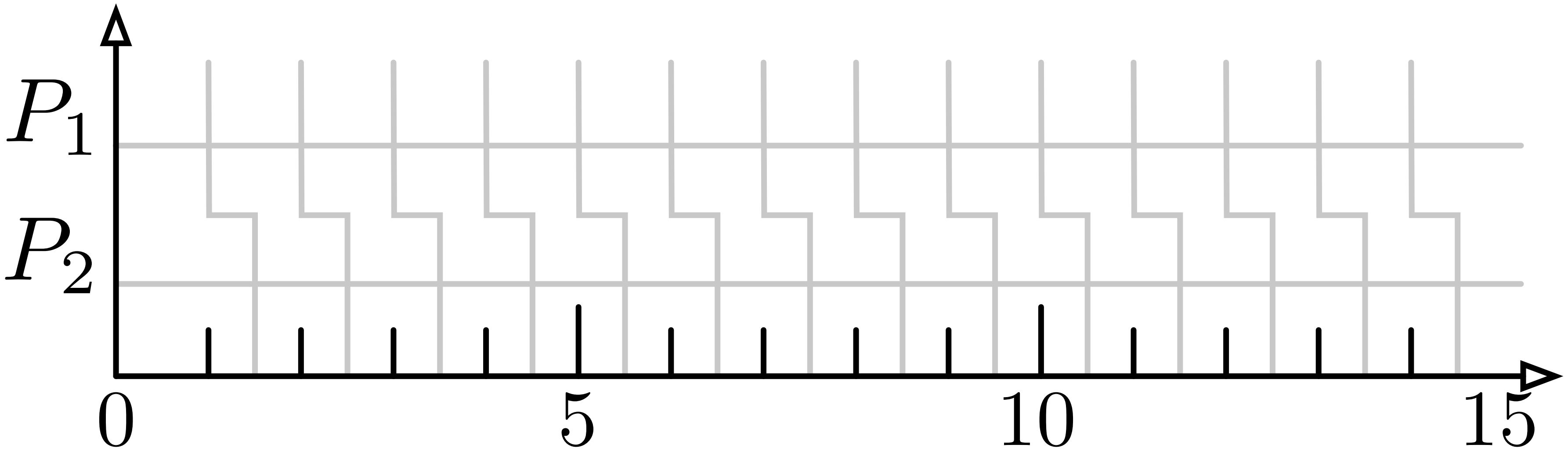
➡ **Contention** at quantum boundary!



Quantum Alignment

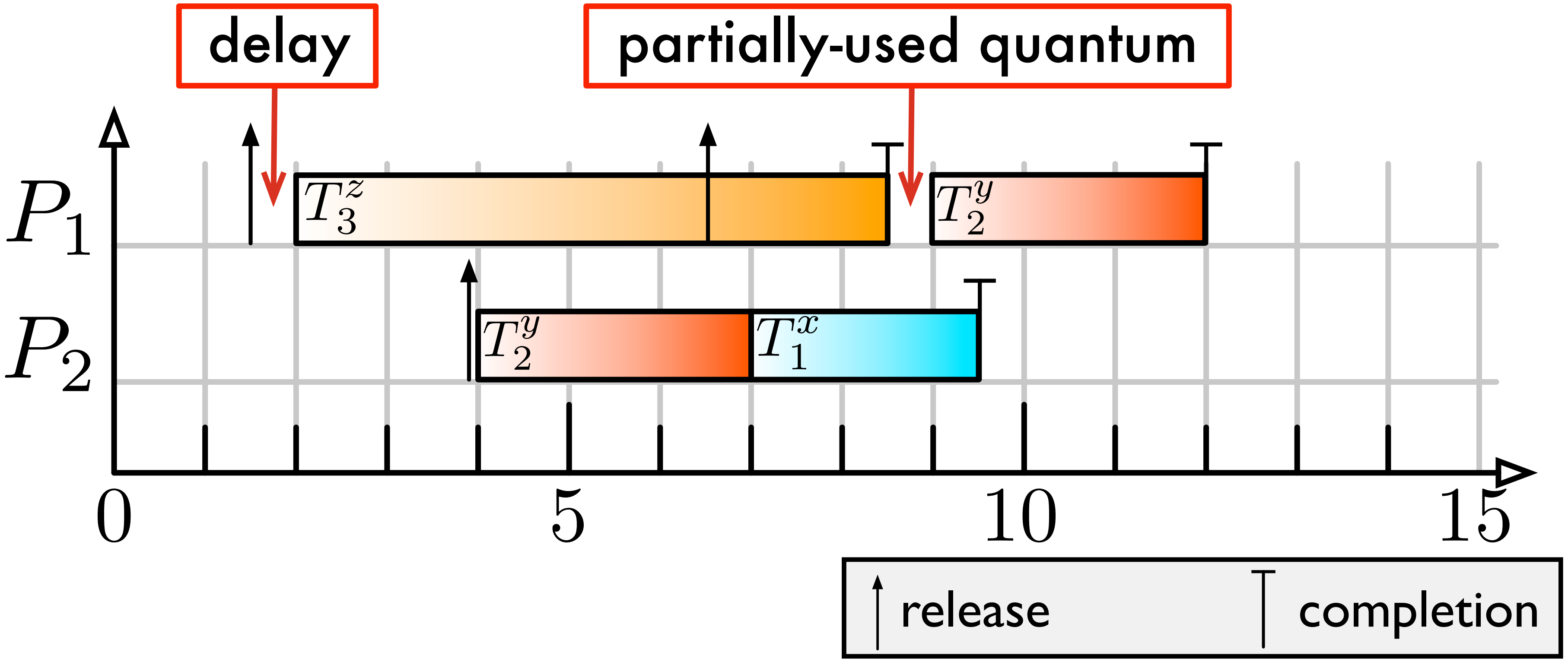
Staggered

- ➡ Ticks spread out across quantum.
- ➡ **Reduced** bus and lock contention.
- ➡ Additional **latency**.



Aligned

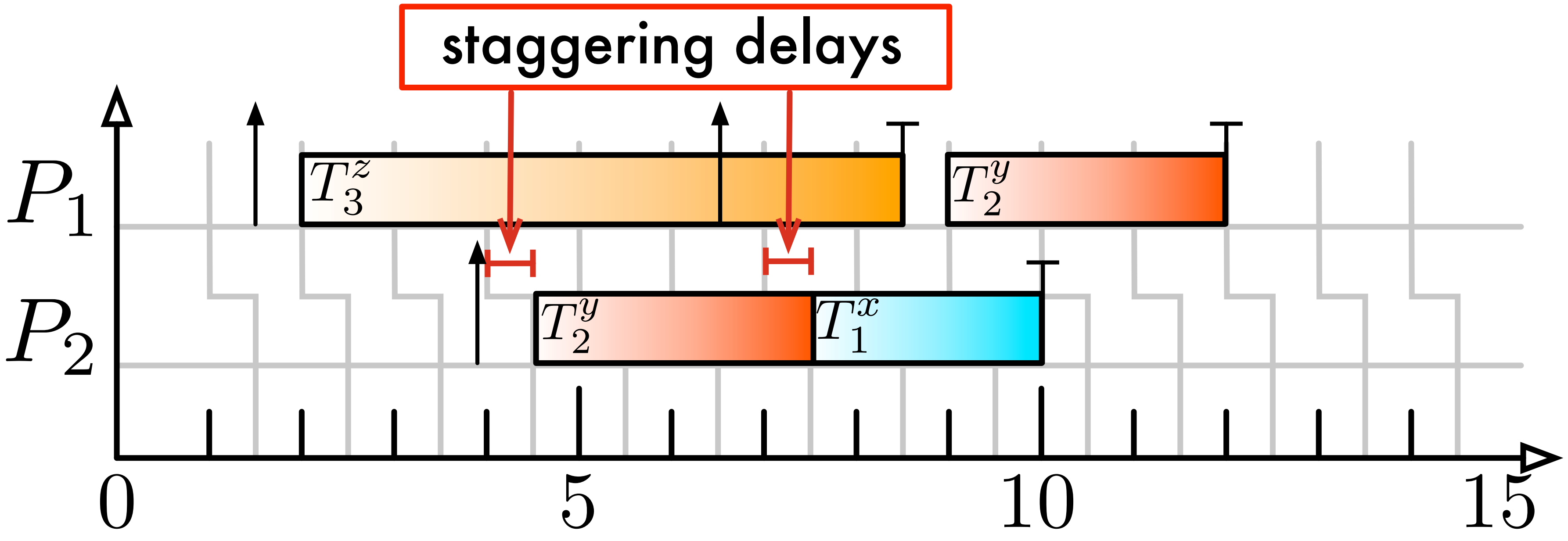
- ➡ Tick **synchronized** across processors.
- ➡ **Contention** at quantum boundary!



Quantum Alignment

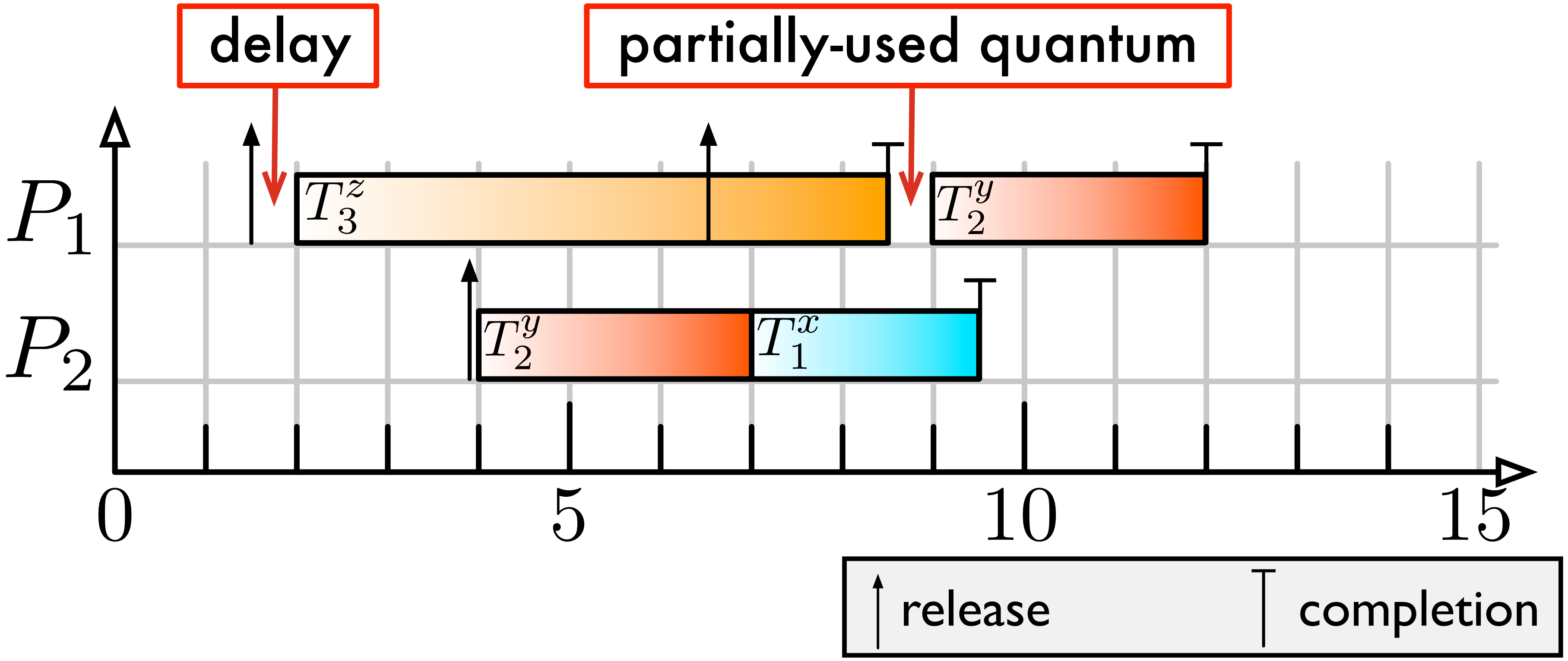
Staggered

- ➡ Ticks spread out across quantum.
- ➡ **Reduced** bus and lock contention.
- ➡ Additional **latency**.



Aligned

- ➡ Tick **synchronized** across processors.
- ➡ **Contention** at quantum boundary!



Quantum Alignment

Staggered

➡ Ticks spread out across quantum.

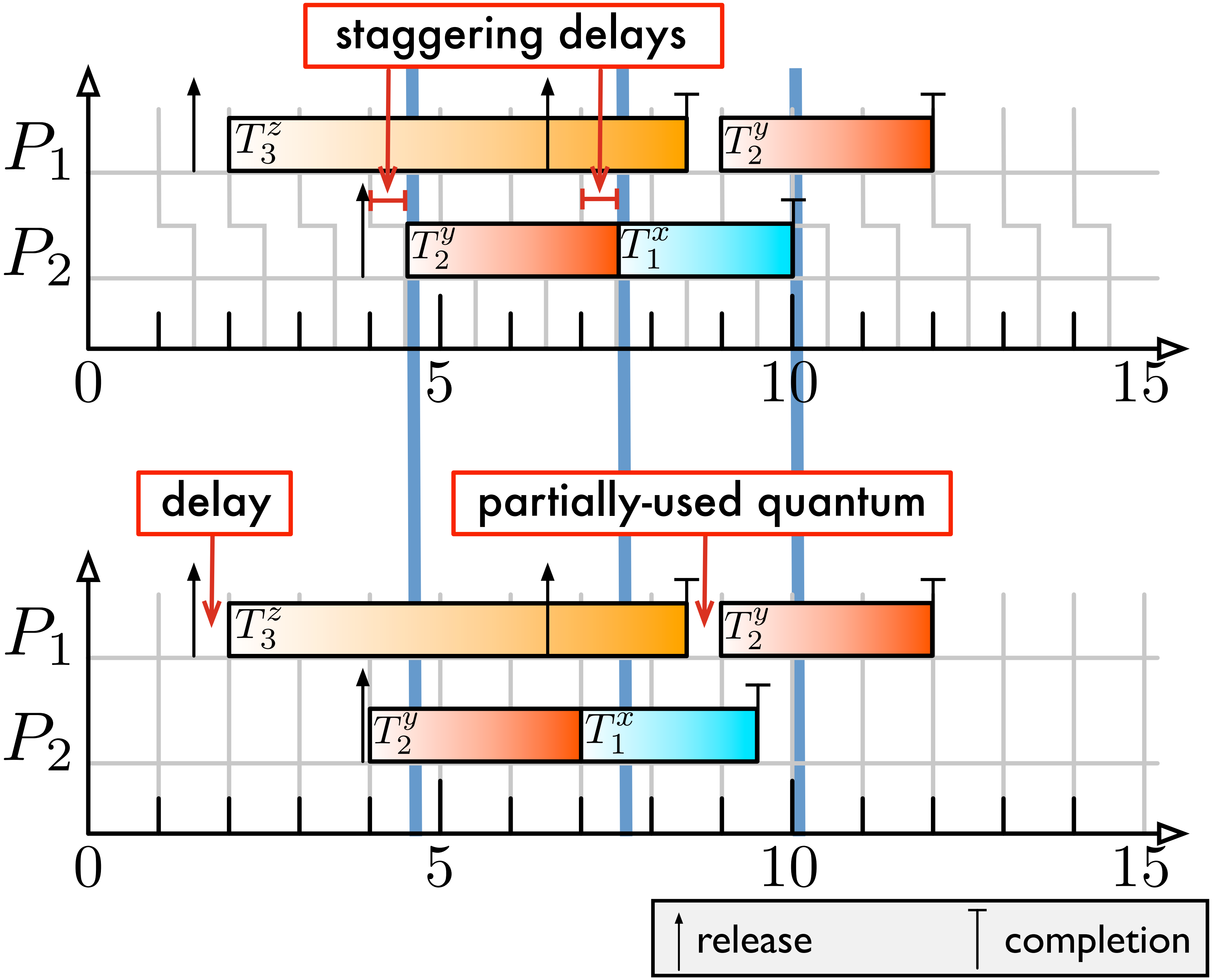
➡ **Reduced** bus and lock contention.

➡ Additional **latency**.

Aligned

➡ Tick **synchronized** across processors.

➡ **Contention** at quantum boundary!



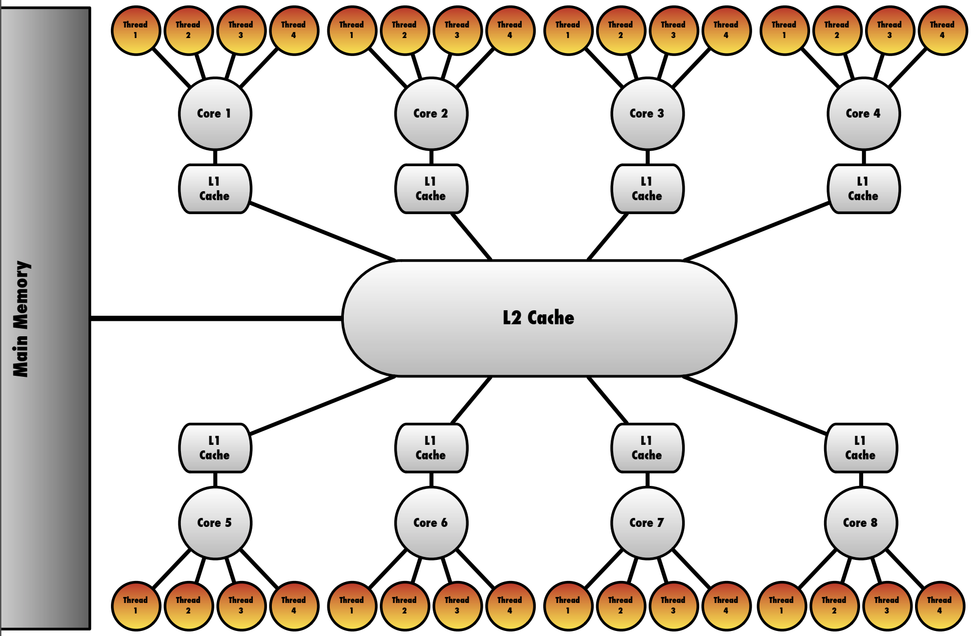
staggering delays

delay

partially-used quantum

Interrupt Handling

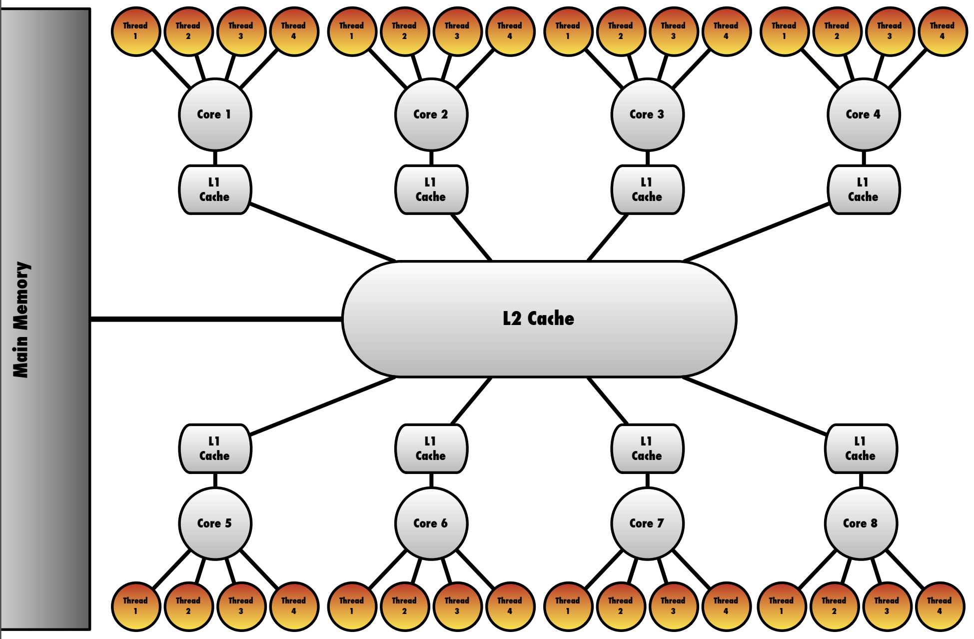
Interrupt Handling



Global interrupt handling.

- ➡ Job releases triggered by **interrupts**.
- ➡ Interrupts may fire **on any processor**.
- ➡ Jobs may execute **on any processor**.
- ➡ Thus, in the worst case, a job may be **delayed by each interrupt**.

Interrupt Handling



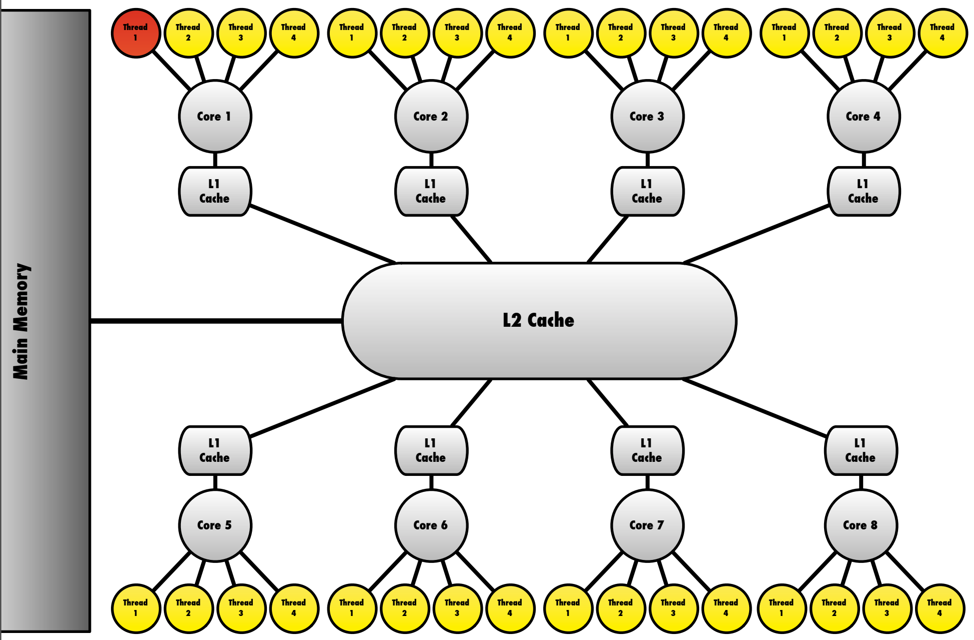
Global interrupt handling.

➡ Job releases triggered by **interrupts**.

➡ Interrupts may fire **on any processor**.

➡ Jobs may execute **on any processor**.

➡ Thus, in the worst case, a job may be **delayed by each interrupt**.



Dedicated interrupt handling.

➡ **Only one processor** services interrupts.

➡ Jobs may execute **on other processors**.

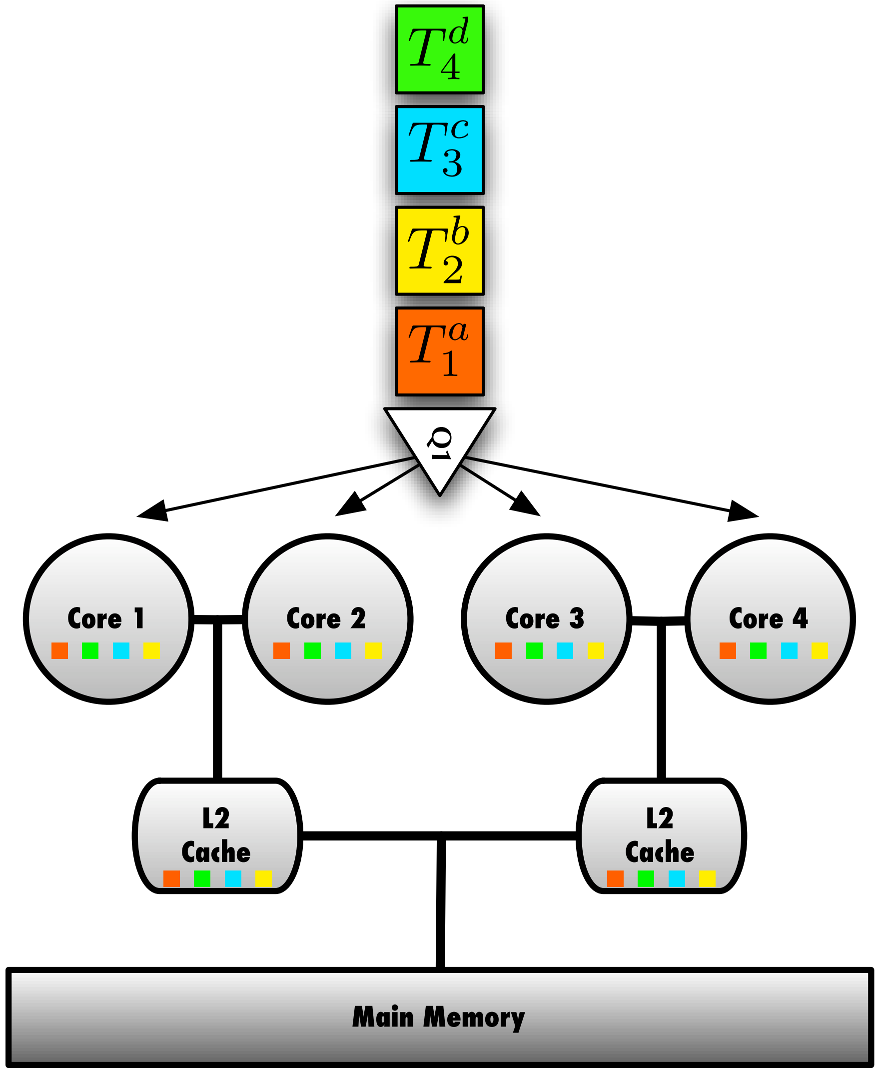
➡ Jobs are not delayed by release interrupts.

➡ Well-known technique; used in the **Spring** kernel (Stankovic and Ramamritham, 1991).

➡ How does it affect **schedulability**?

J.A. Stankovic and K. Ramamritham (1991), The Spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72.

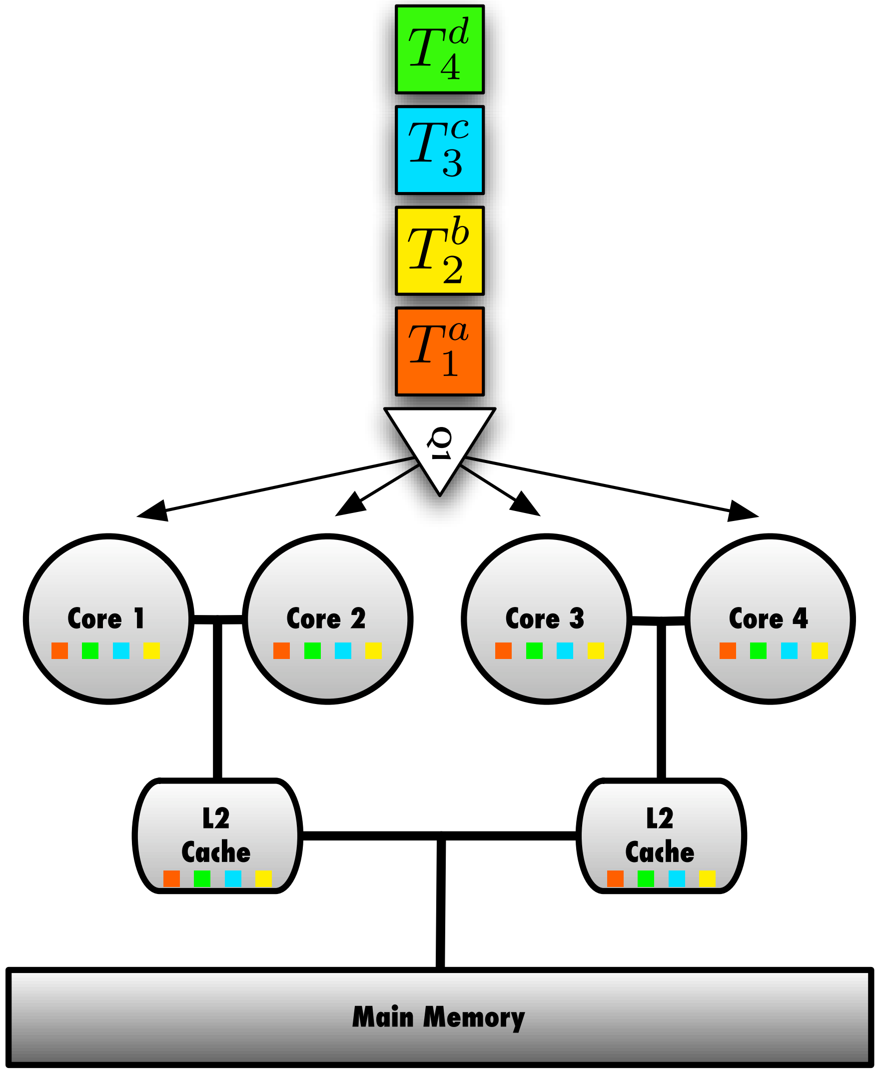
Ready Queue


T_4^d
T_3^c
T_2^b
T_1^a
Q1
Core 1
Core 2
Core 3
Core 4
L2 Cache
L2 Cache
Main Memory

Ready Queue

Globally-shared priority queue.

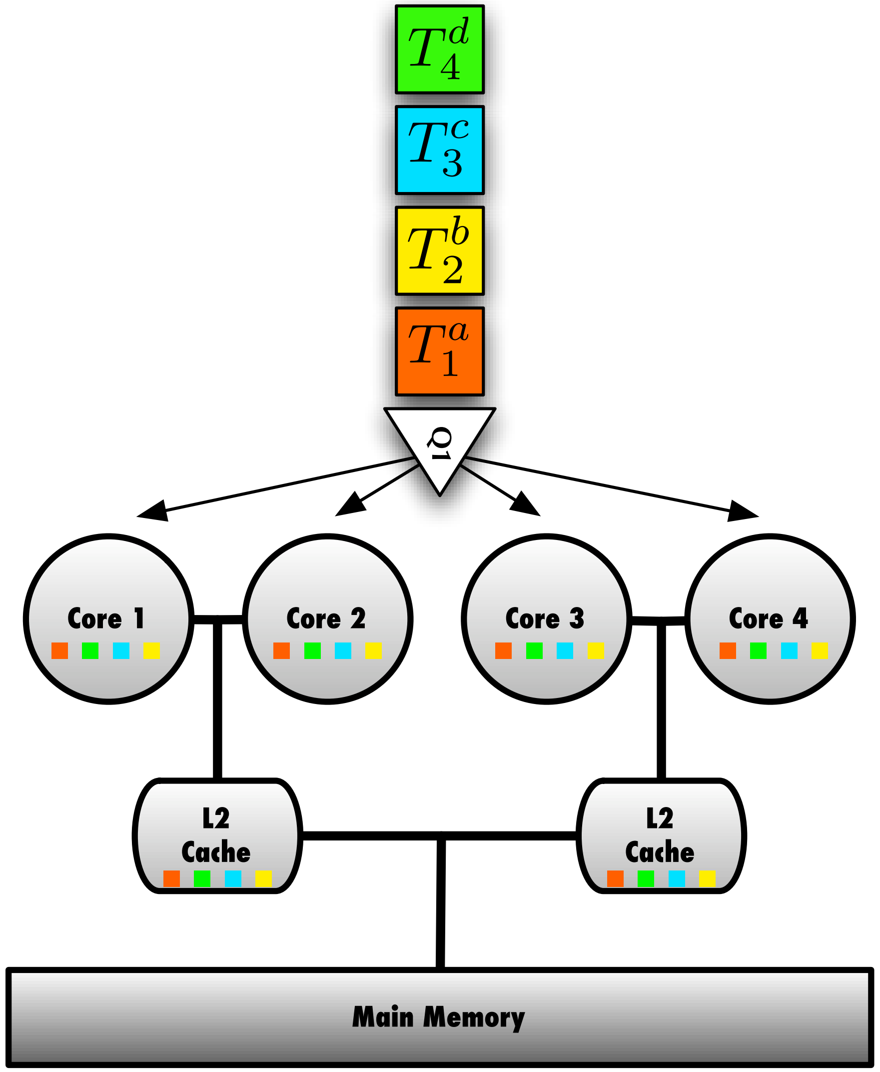
- ➡ Problem: **hyper-period boundaries**.
- ➡ Problem: **lock contention**.
- ➡ Problem: **bus contention**.



Ready Queue

Globally-shared priority queue.

- ➡ Problem: **hyper-period boundaries**.
- ➡ Problem: **lock contention**.
- ➡ Problem: **bus contention**.



Requirements.

- ➡ **Mergeable** priority queue: release *n* jobs in O(log *n*) time.
- ➡ **Parallel** enqueue / dequeue operations.
- ➡ Mostly **cache-local** data structures.

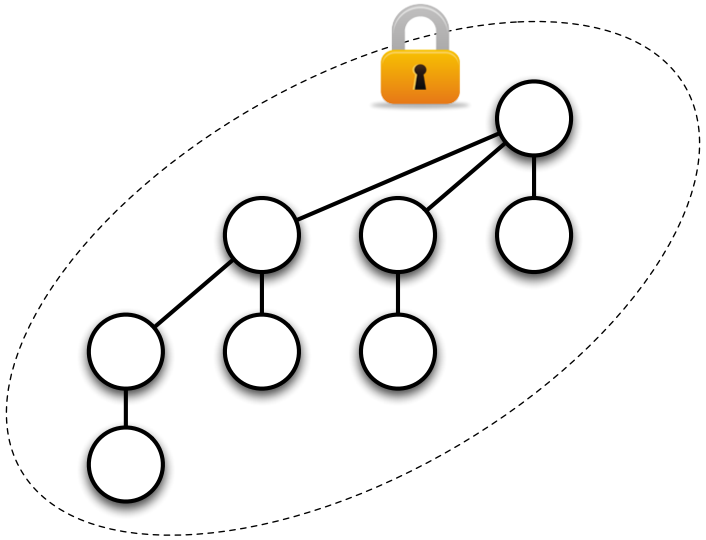
Ready Queue

Globally-shared priority queue.

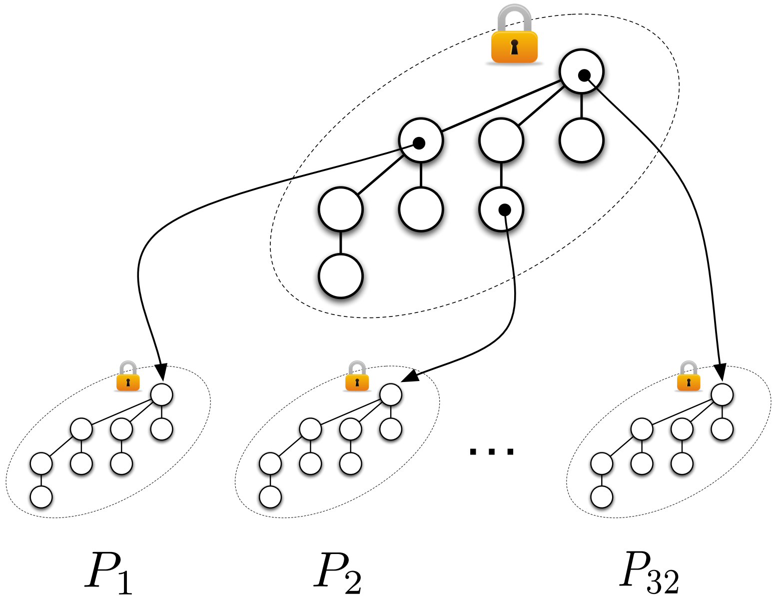
- ➡ Problem: **hyper-period boundaries**.
- ➡ Problem: **lock contention**.
- ➡ Problem: **bus contention**.

In this study, we consider three queue implementations.

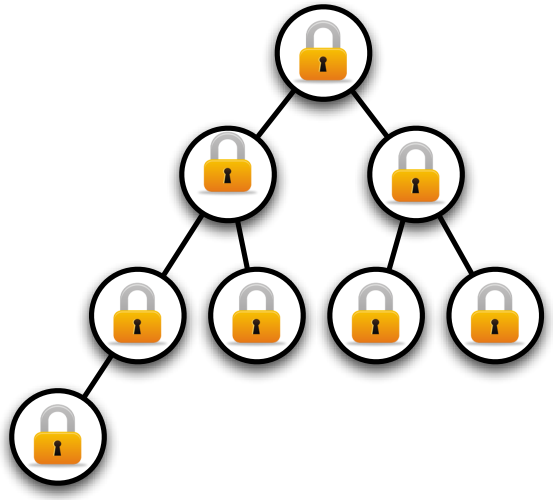
Coarse-Grained Heap



Hierarchical Heaps



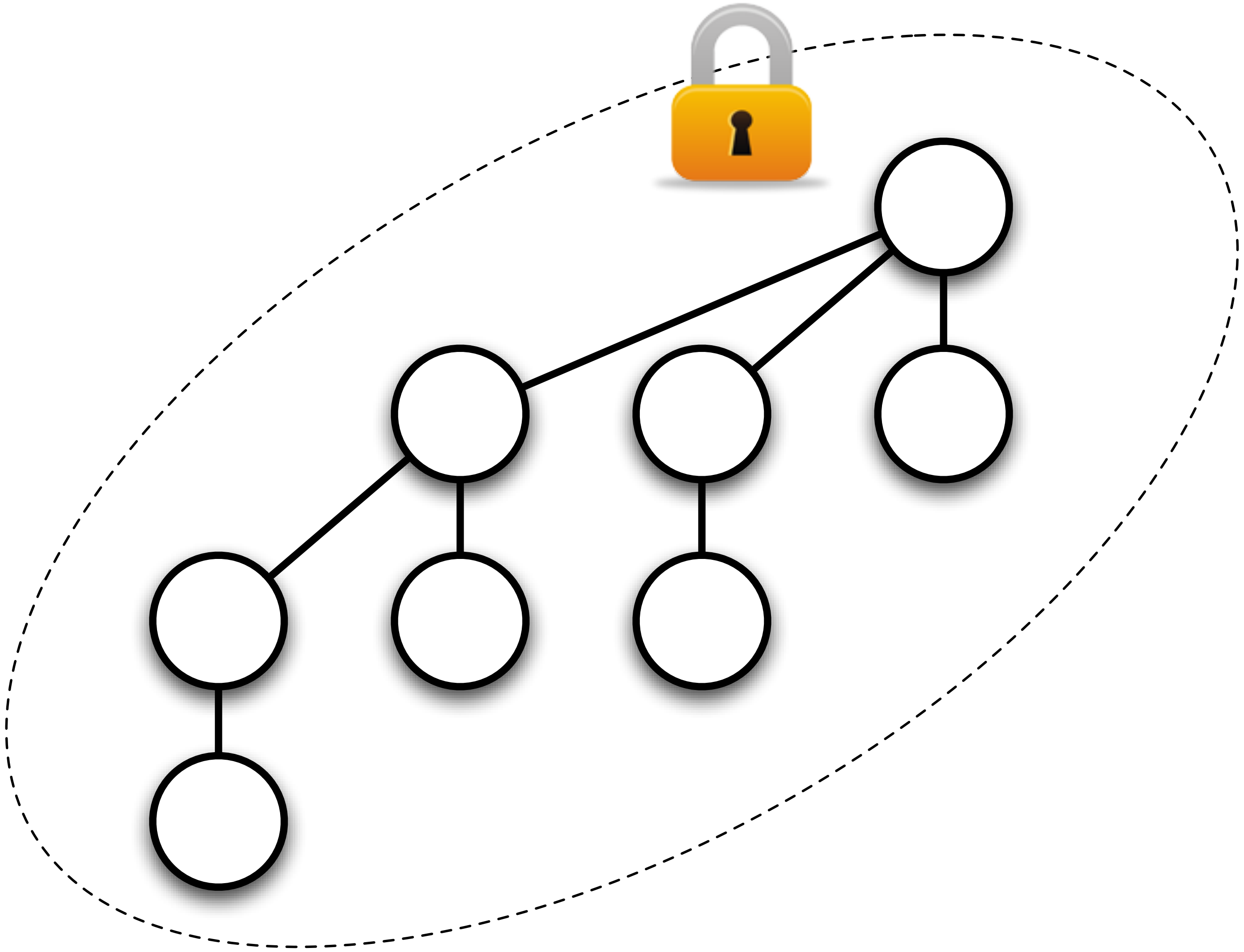
Fine-Grained Heap



Ready Queue: Coarse-Grained Heap

Binomial heap + single lock.

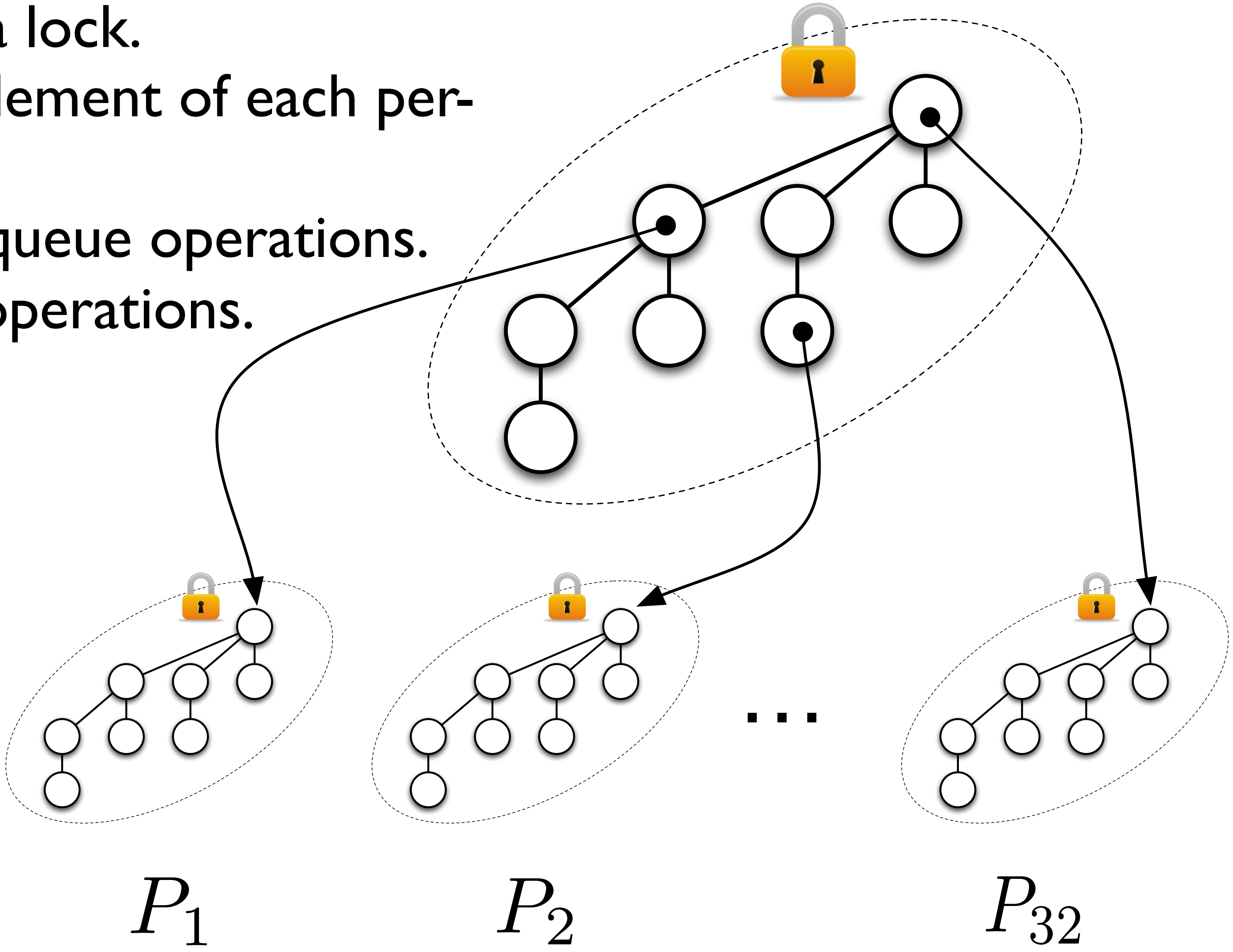
- ➡ Lock used to synchronize all G-EDF state.
- ➡ **Mergeable** queue.
- ➡ No parallel updates.
- ➡ No cache-local updates.
- ➡ Low locking overhead (only single lock acquisition).



Ready Queue: Hierarchical Heaps

Per-processor queues + master queue.

- ➡ Each queue protected by a lock.
- ➡ Master queue holds min element of each per-processor queue.
- ➡ **Global, sequential** dequeue operations.
- ➡ **Mostly-local** enqueue operations.



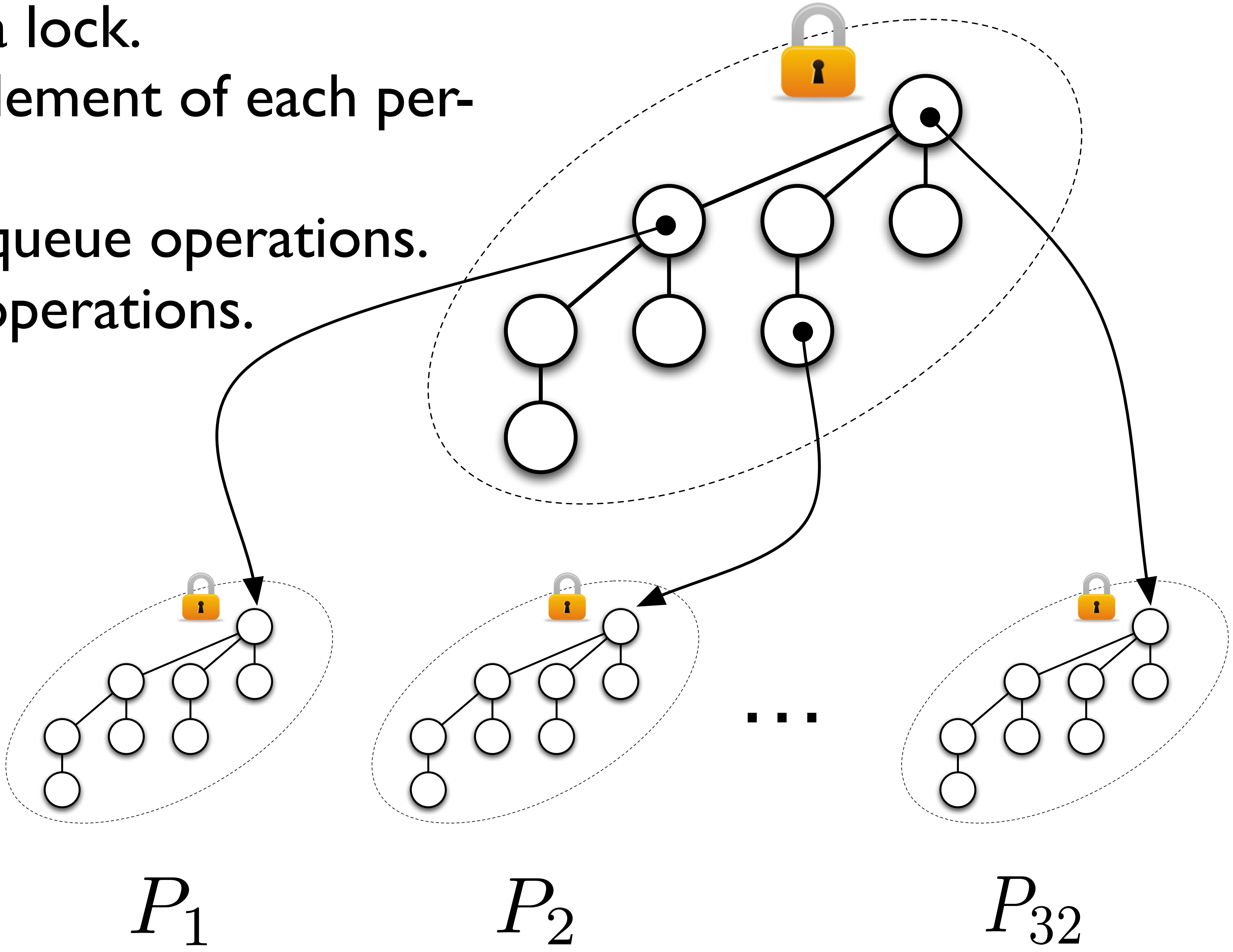
Ready Queue: Hierarchical Heaps

Per-processor queues + master queue.

- ➡ Each queue protected by a lock.
- ➡ Master queue holds min element of each per-processor queue.
- ➡ **Global, sequential** dequeue operations.
- ➡ **Mostly-local** enqueue operations.

Locking.

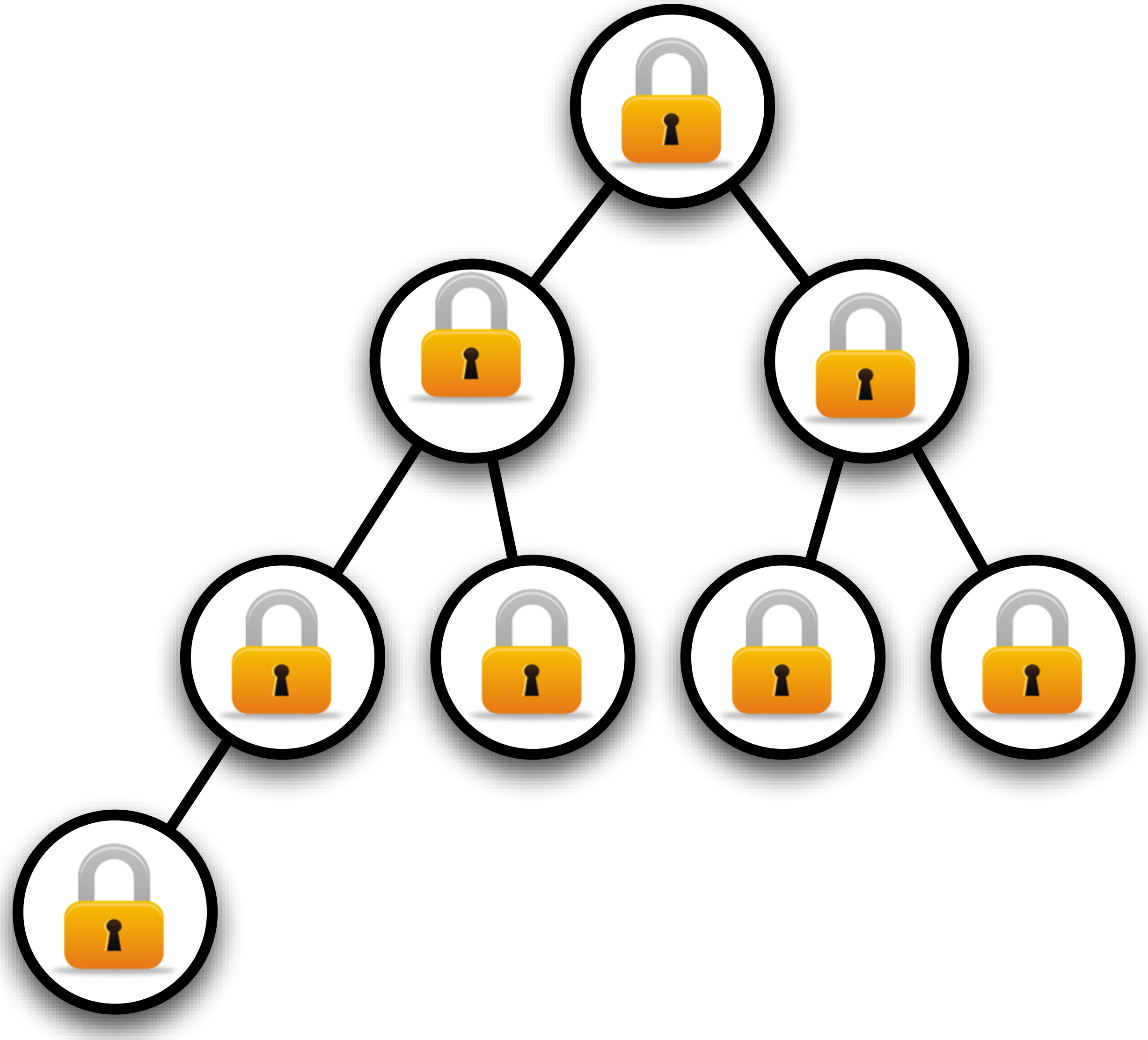
- ➡ Dequeue: top-down.
- ➡ Enqueue: bottom-up.
- ➡ Enqueue may have to drop lock, retry.
- ➡ Additional complexity wrt. dequeue (see paper).
- ➡ Bottom line: **expensive**.



Ready Queue: Fine-Grained Heap

Parallel binary heap.

- ➡ One lock per heap node.
- ➡ Proposed by Hunt et al. (1996).
- ➡ **Not mergeable**.
- ➡ **Parallel enqueue / dequeue**.
- ➡ **No cache-local data**.



Hunt et al. (1996), An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157.

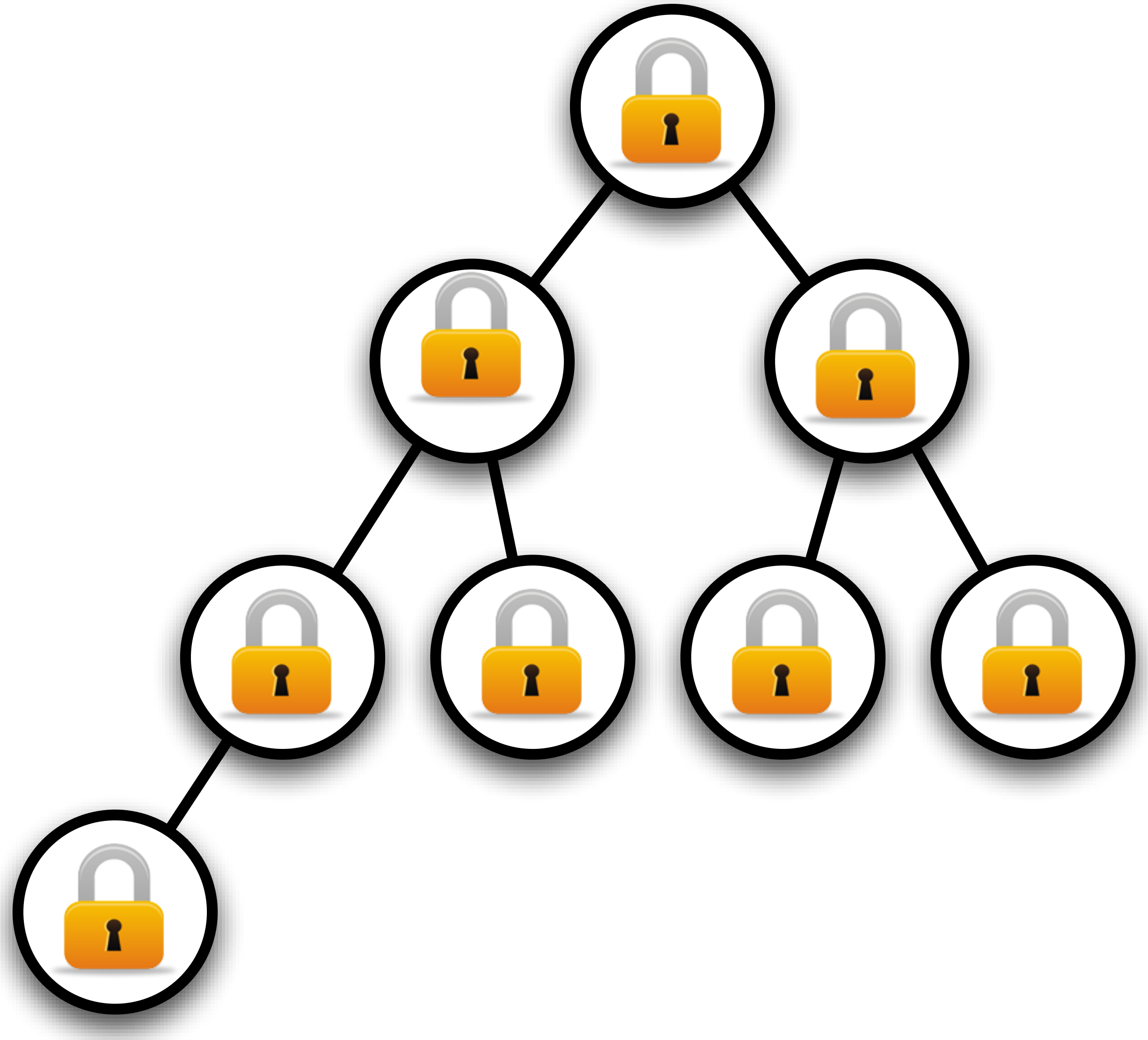
Ready Queue: Fine-Grained Heap

Parallel binary heap.

- ➡ One lock per heap node.
- ➡ Proposed by Hunt et al. (1996).
- ➡ **Not mergeable**.
- ➡ **Parallel enqueue / dequeue**.
- ➡ **No cache-local data**.

Locking.

- ➡ Many lock acquisitions.
- ➡ Atomic **peek+dequeue** operation needed to check for preemptions.



Hunt et al. (1996), An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157.

Additional Components

Release queue.

- ➡ Support mergeable queues.
- ➡ Support dedicated interrupt handling.

Job-to-processor mapping.

- ➡ Quickly determine whether preemption is required.
- ➡ Avoid unnecessary preemptions.
- ➡ Used to linearize concurrent scheduling decisions.

Implementation in ***LITMUS***^{RT}

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

UNC's Linux patch.

- ➡ Used in several previous studies.
- ➡ On-going development.
- ➡ Currently, based off of Linux 2.6.24.

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

UNC's Linux patch.

- ➡ Used in several previous studies.
- ➡ On-going development.
- ➡ Currently, based off of Linux 2.6.24.

Scheduler Plugin API.

- ➡ `scheduler_tick()`
- ➡ `schedule()`
- ➡ `release_jobs()`

Considered G-EDF Variants

Name	**Ready Q**	**Scheduling**	**Interrupts**

Considered G-EDF Variants

Name	**Ready Q**	**Scheduling**	**Interrupts**
CEm	coarse-grained	event-driven	global
CQm	coarse-grained	quantum (aligned)	global
S-CQm	coarse-grained	quantum (staggered)	global
HEm	hierarchical	event-driven	global
FEm	fine-grained	event-driven	global

Co nts

Baseline from
(Brandenburg et al., 2008)

Name	Ready Q	Scheduling	Interrupts
CEm	coarse-grained	event-driven	global
CQm	coarse-grained	quantum (aligned)	global
S-CQm	coarse-grained	quantum (staggered)	global
HEm	hierarchical	event-driven	global
FEm	fine-grained	event-driven	global

No fine-grained heaps + quantum-driven scheduling.

(Parallel updates not beneficial due to quantum barrier.)

Name	Ready Q	Scheduling	Interrupts
CEm	coarse-grained	event-driven	global
CQm	coarse-grained	quantum (aligned)	global
S-CQm	coarse-grained	quantum (staggered)	global
HEm	hierarchical	event-driven	global
FEm	fine-grained	event-driven	global

Considered G-EDF Variants

Name	**Ready Q**	**Scheduling**	**Interrupts**
CEm	coarse-grained	event-driven	global
CQm	coarse-grained	quantum (aligned)	global
S-CQm	coarse-grained	quantum (staggered)	global
HEm	hierarchical	event-driven	global
FEm	fine-grained	event-driven	global
CE1	coarse-grained	event-driven	dedicated
CQ1	coarse-grained	quantum (aligned)	dedicated
S-CQ1	coarse-grained	quantum (staggered)	dedicated
FE1	fine-grained	event-driven	dedicated

No hierarchical heaps + dedicated interrupt handling.

(Hierarchical heaps not beneficial if only one proc. enqueues.)

Name	Ready Q	Scheduling	Interrupts
CEm	coarse-grained	event-driven	global
CQm	coarse-grained	quantum (aligned)	global
S-CQm	coarse-grained	quantum (staggered)	global
HEm	hierarchical	event-driven	global
FEm	fine-grained	event-driven	global
CE1	coarse-grained	event-driven	dedicated
CQ1	coarse-grained	quantum (aligned)	dedicated
S-CQ1	coarse-grained	quantum (staggered)	dedicated
FE1	fine-grained	event-driven	dedicated

Schedulability Study

Objective

Compare the discussed implementations
in terms of the ratio of randomly-generated task sets
that can be shown to be schedulable
***under consideration of system overheads**.*

Scheduling Overheads

Scheduling Overheads

Release overhead.

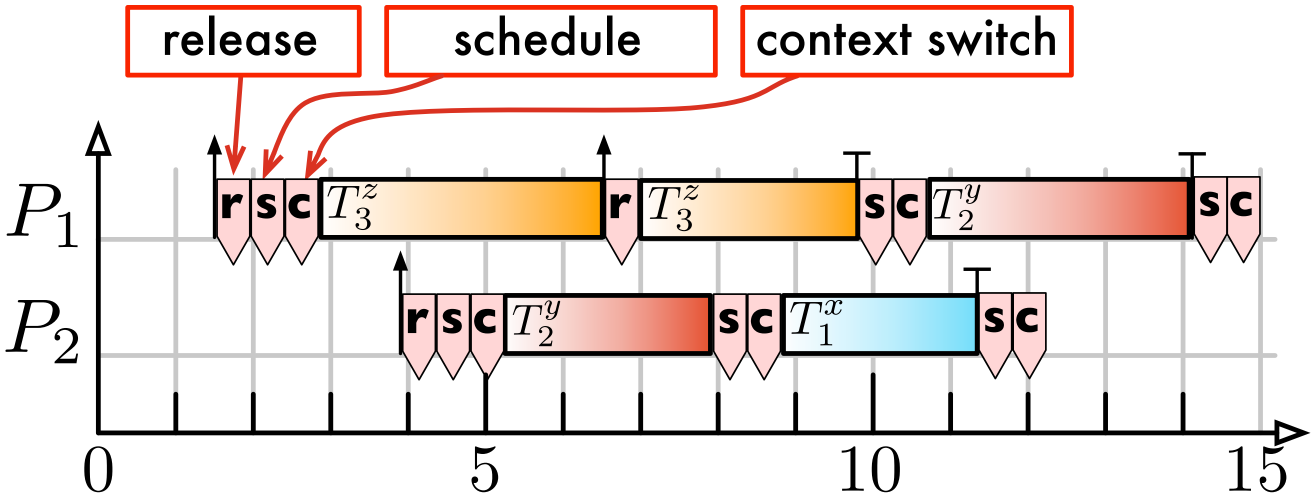
➡ The cost of a one-shot timer interrupt.

Scheduling overhead.

➡ Selecting the next job to run.

Context switch overhead.

➡ Changing address space.



Scheduling Overheads

Release overhead.

➡ The cost of a one-shot timer interrupt.

Scheduling overhead.

➡ Selecting the next job to run.

Context switch overhead.

➡ Changing address space.

Tick overhead.

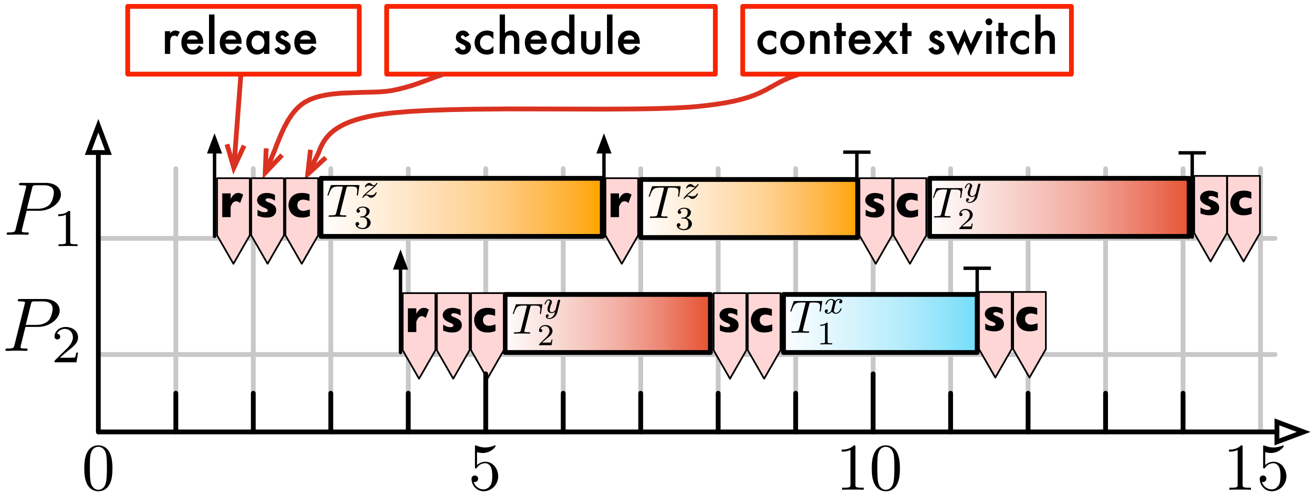
➡ Cost of a periodic timer interrupt.

➡ Beginning of a new quantum.

Preemption and migration overhead.

➡ Loss of cache affinity.

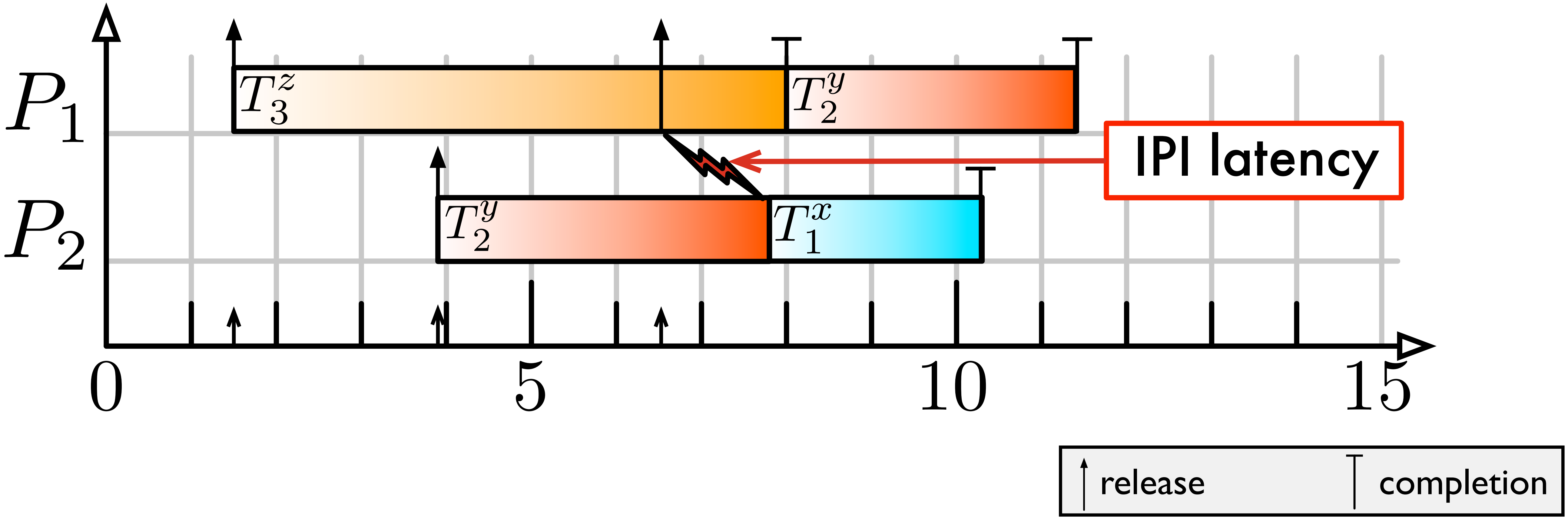
➡ Known from (Brandenburg et al., 2008).



IPI Latency

Inter-processor interrupts (IPIs).

- ➡ Interrupt may be processed by a processor different from the one that will schedule a newly-arrived job.
- ➡ Requires notification of remote processor.
- ➡ **Event-based scheduling incurs added latency**.



Test Platform

— SUN UltraSPARC T1 "Niagara"

LITMUS^{RT}
- ➡ UNC's Linux-based Real-Time Testbed

Sun UltraSPARC T1 "Niagara"
- ➡ 8 cores, 4 HW threads per core = 32 logical processors.
- ➡ 3 MB shared L2 cache

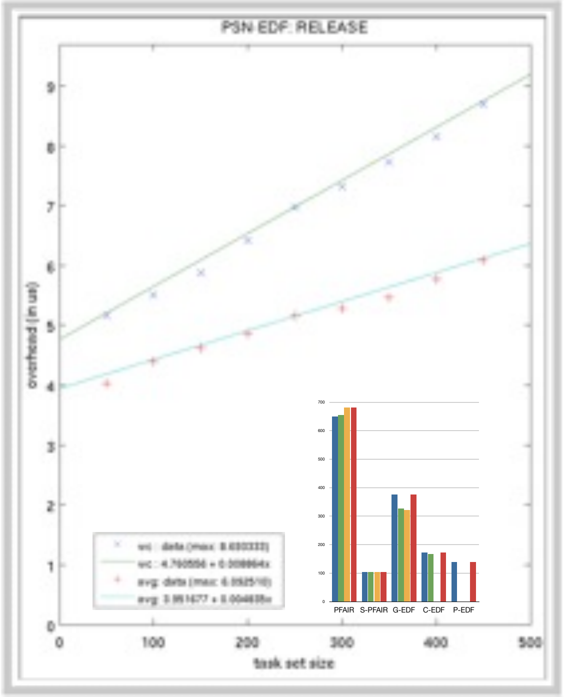
Test Platform

LITMUS^{RT}

- ➡ UNC's Linux-based Real-Time Testbed

Sun UltraSPARC T1 "Niagara"

- ➡ 8 cores, 4 HW threads per core = 32 logical processors.
- ➡ 3 MB shared L2 cache



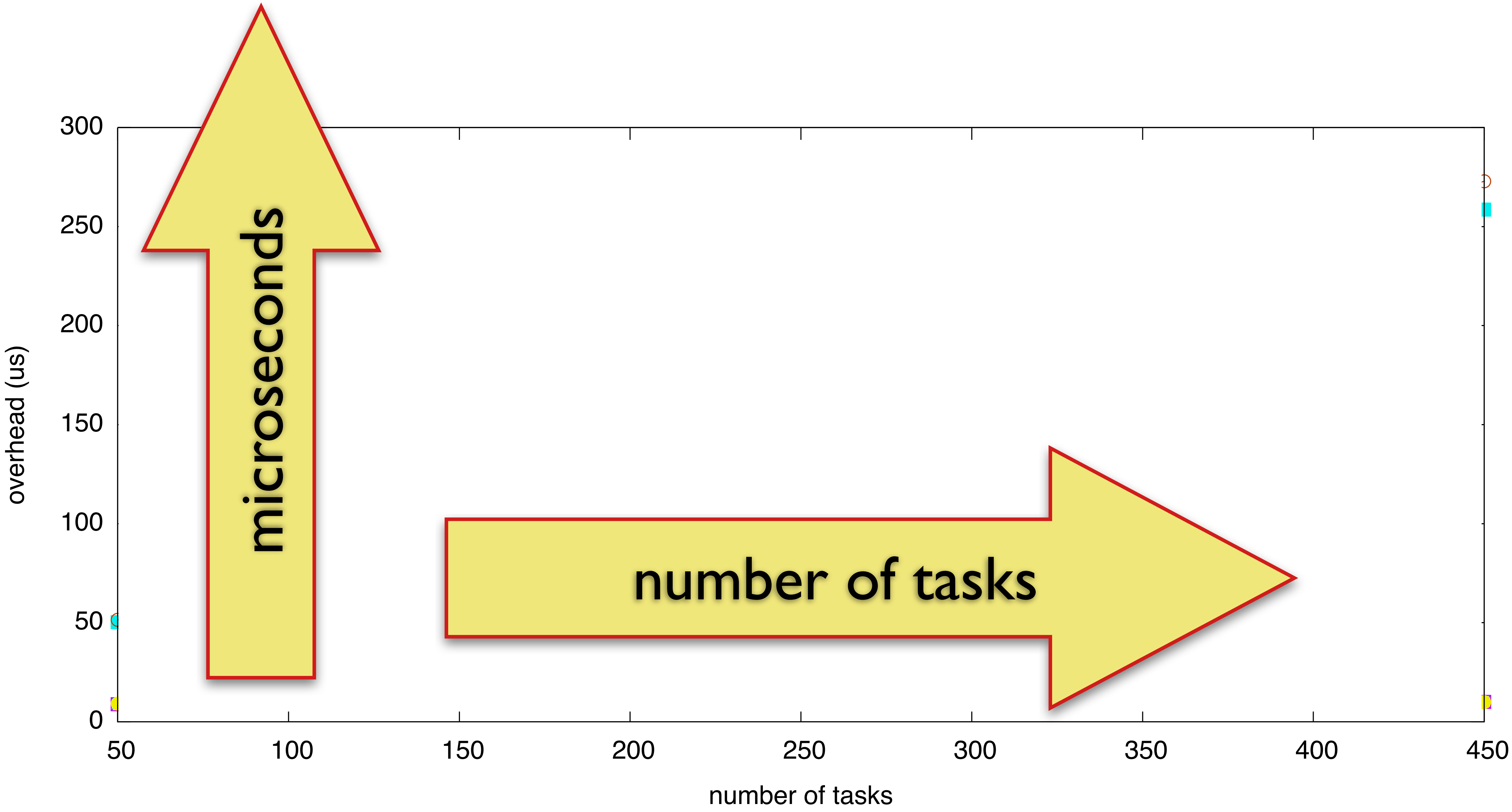
Overheads

- ➡ Traced overheads under each of the plugins.
- ➡ Collected more than 640,000,000 samples (total).
- ➡ Computed worst-case and average-case overheads.
- ➡ Over 20 graphs; see online version.

Outliers

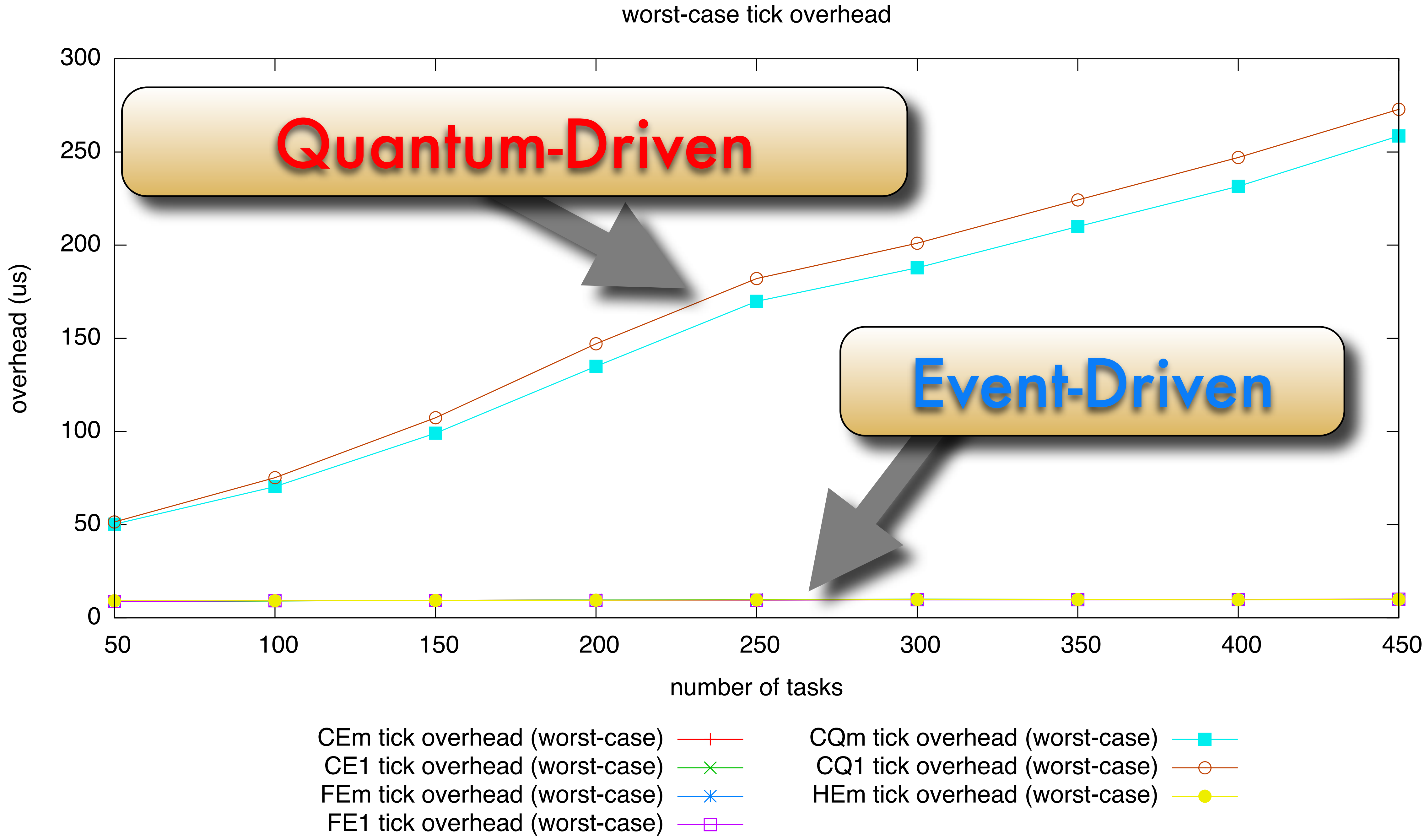
- ➡ Removed top 1% of samples to discard outliers.

Example: Tick Overhead

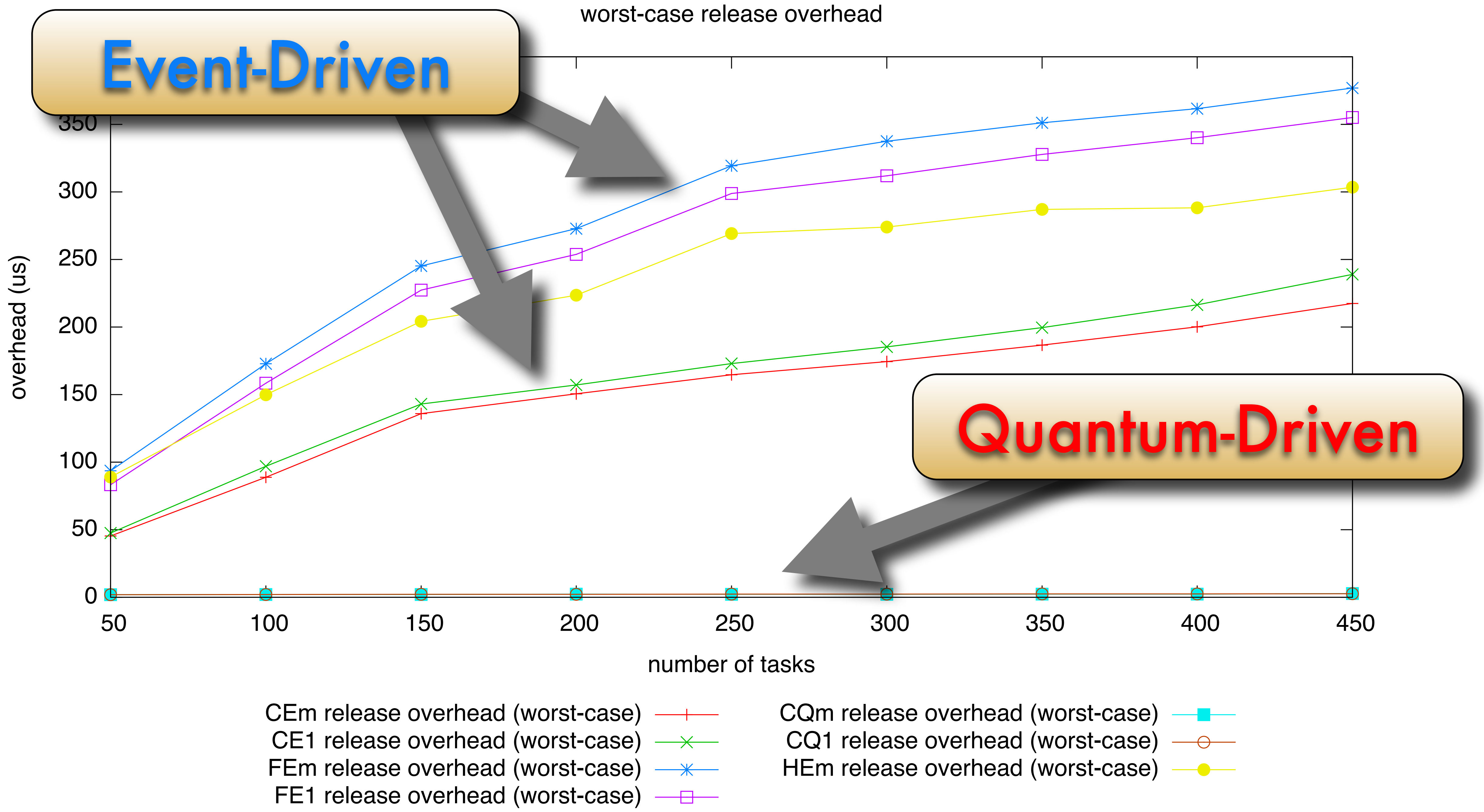


“Higher is worse.”

Example: Tick Overhead



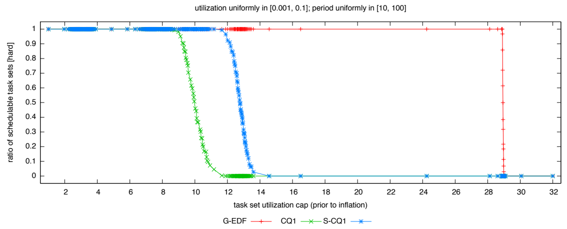
Example: Release Overhead

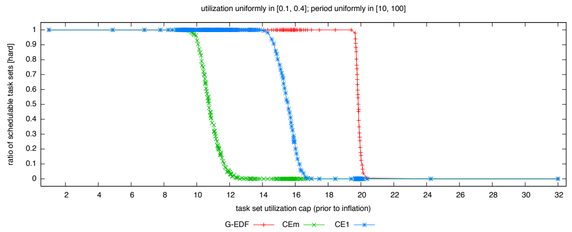


Event-Driven

Quantum-Driven

Study Setup

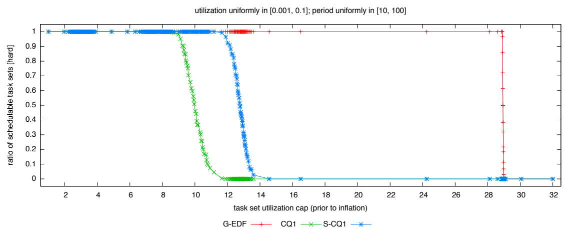


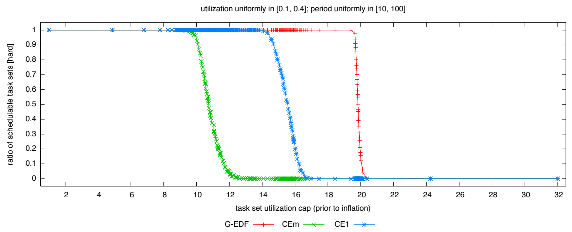


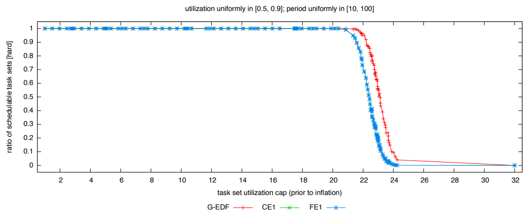
Methodology.

- ➡ Randomly generate task set.
- ➡ Apply overheads (for each G-EDF implementation).
- ➡ Test whether task set can be claimed schedulable (for each G-EDF implementation).

Study Setup







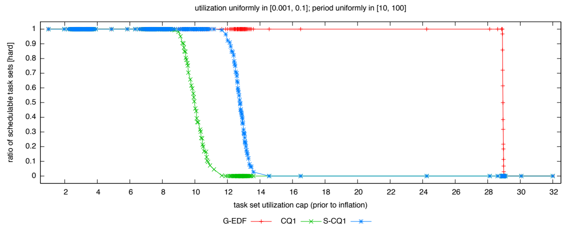
Methodology.

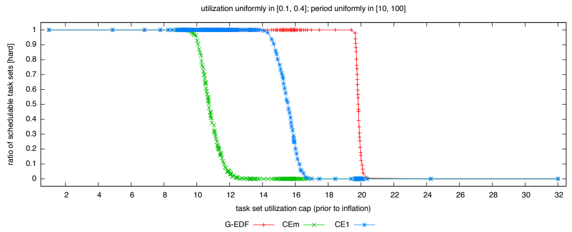
- ➡ Randomly generate task set.
- ➡ Apply overheads (for each G-EDF implementation).
- ➡ Test whether task set can be claimed schedulable (for each G-EDF implementation).

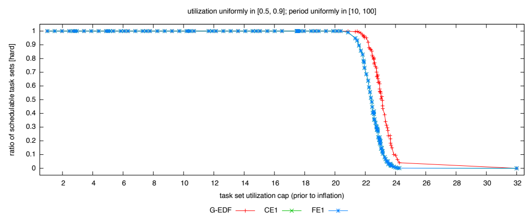
Schedulability.

- ➡ Hard real-time: worst-case overheads, no tardiness.
- ➡ Soft real-time: average-case overheads, bounded tardiness.

Study Setup







Methodology.

- ➡ Randomly generate task set.
- ➡ Apply overheads (for each G-EDF implementation).
- ➡ Test whether task set can be claimed schedulable (for each G-EDF implementation).

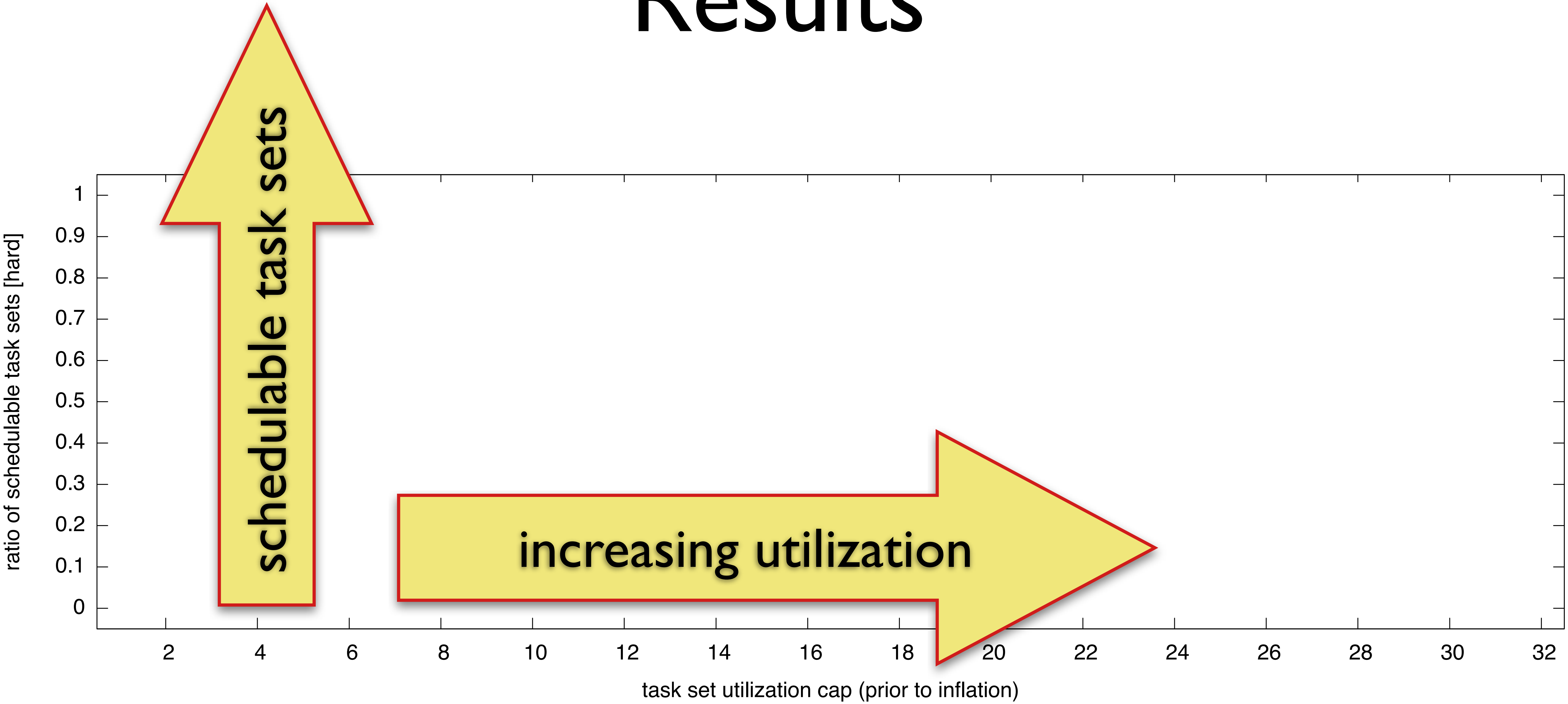
Schedulability.

- ➡ Hard real-time: worst-case overheads, no tardiness.
- ➡ Soft real-time: average-case overheads, bounded tardiness.

Task set generation.

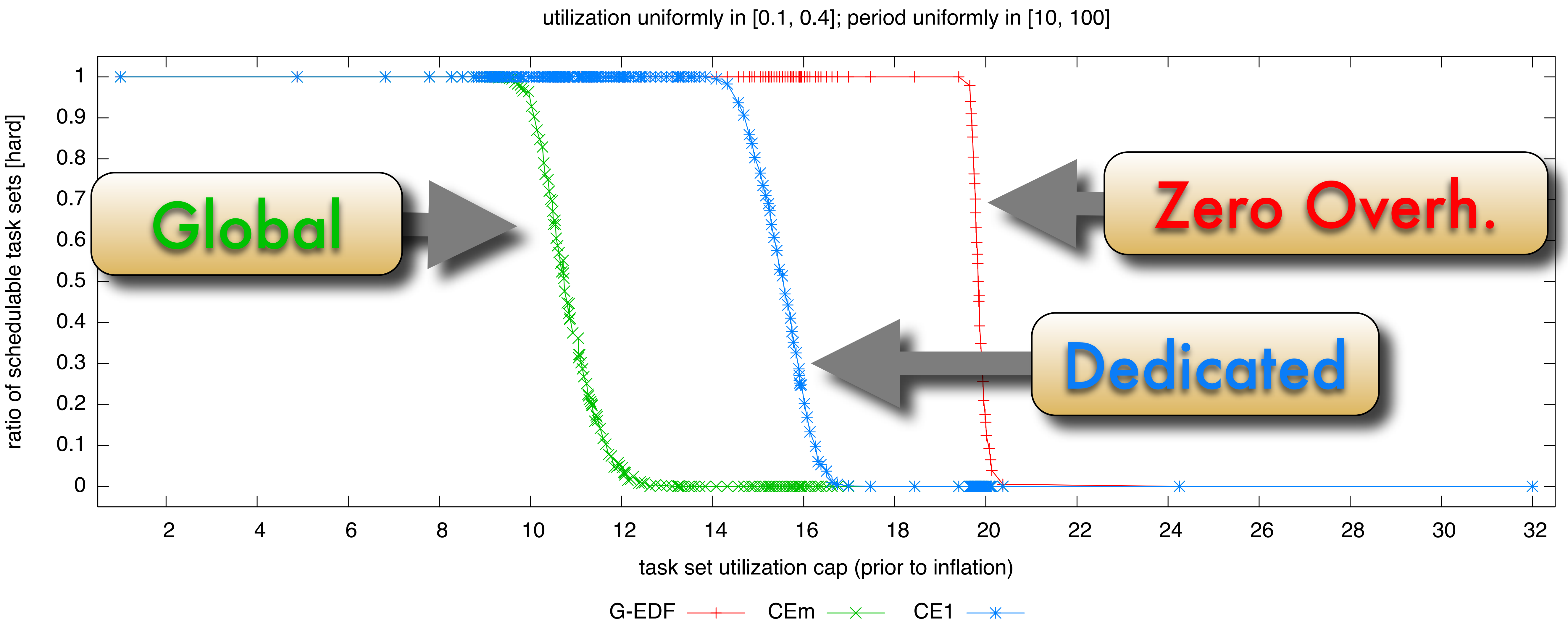
- ➡ Six utilization distributions (uniform and bimodal).
- ➡ Three period distributions (uniform).
- ➡ Over 300 graphs; see online version.

Results



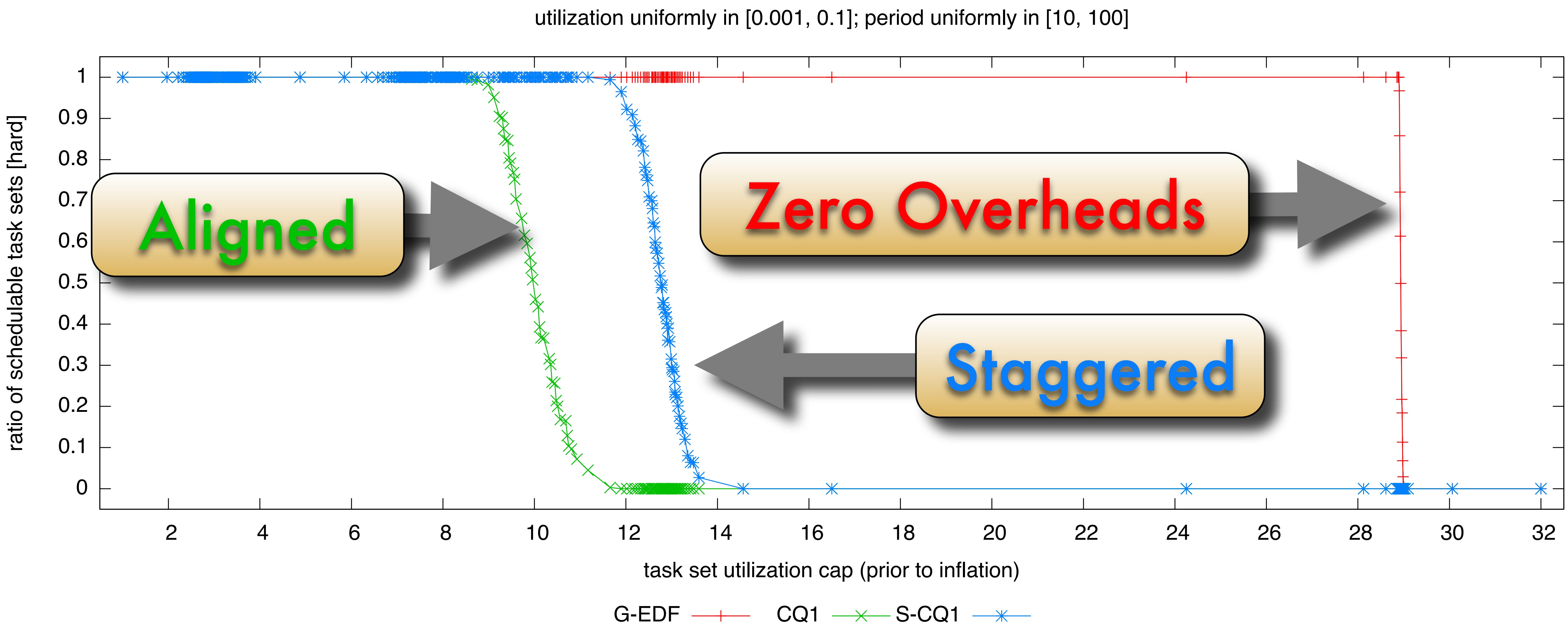
“Higher is better.”

Interrupt Handling



Dedicated interrupt handling
was generally preferable (or no worse).

Quantum Staggering



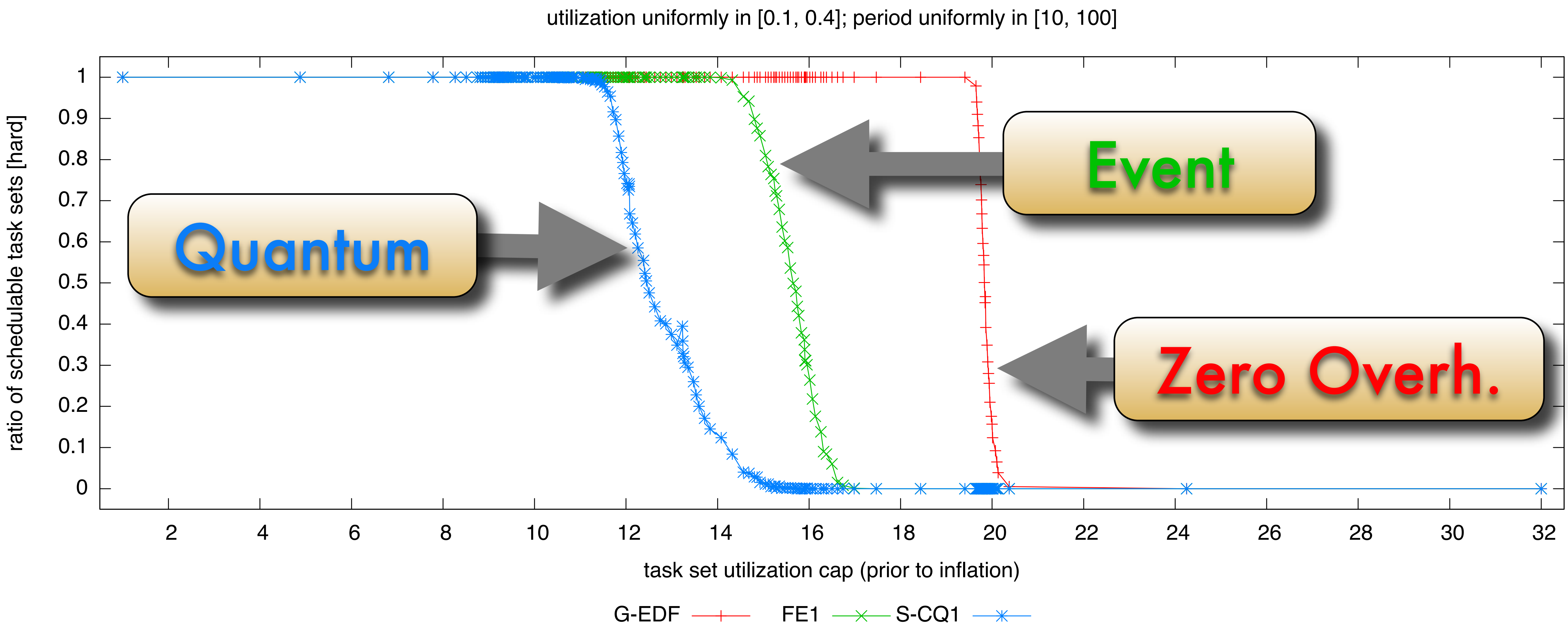
Aligned

Zero Overheads

Staggered

Staggered quanta
were generally preferable (or no worse).

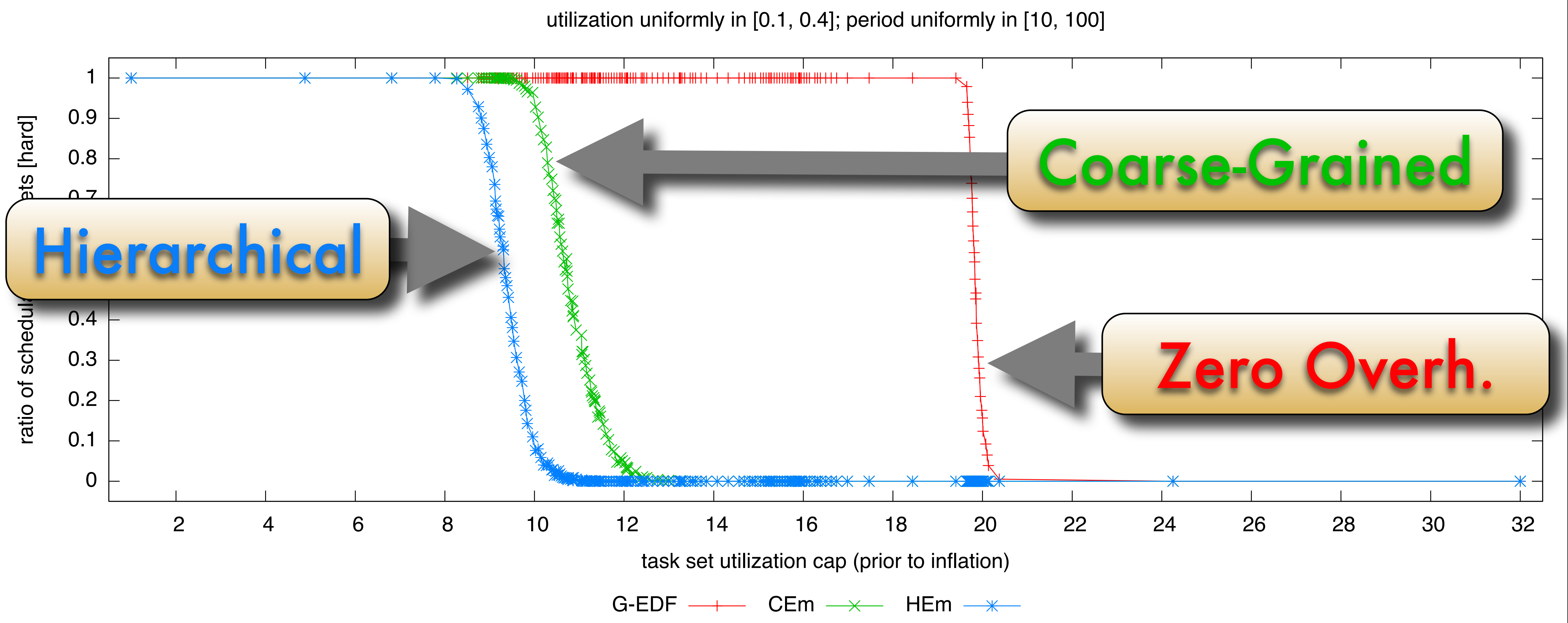
Quantum- vs. Event-Driven



Event-driven scheduling

was preferable in most cases.

Choice of Ready Queue (1)



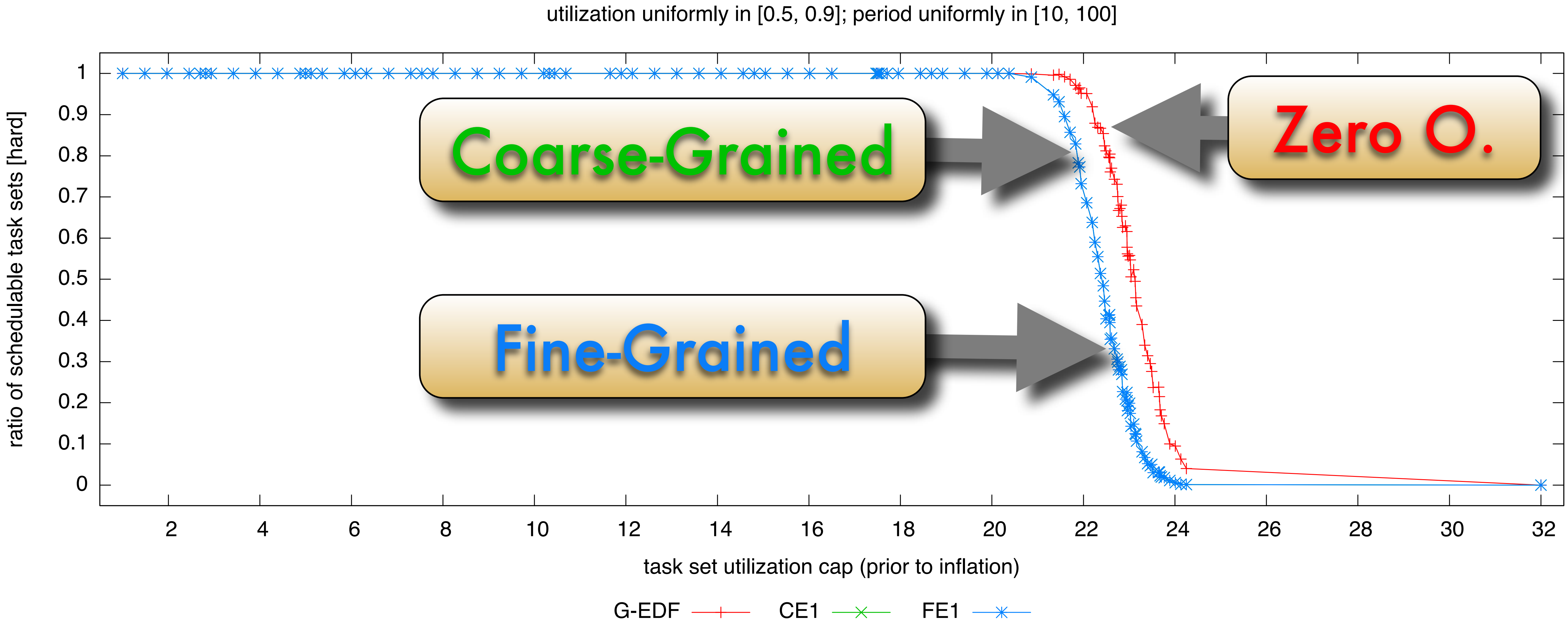
Coarse-Grained

Hierarchical

Zero Overh.

The **coarse-grained ready queue** performed better than the hierarchical queue.

Choice of Ready Queue (II)



The fine-grained ready queue
performed marginally better than the coarse-grained queue
if used together with **dedicated interrupt handling**.

Conclusion

Summary of Results

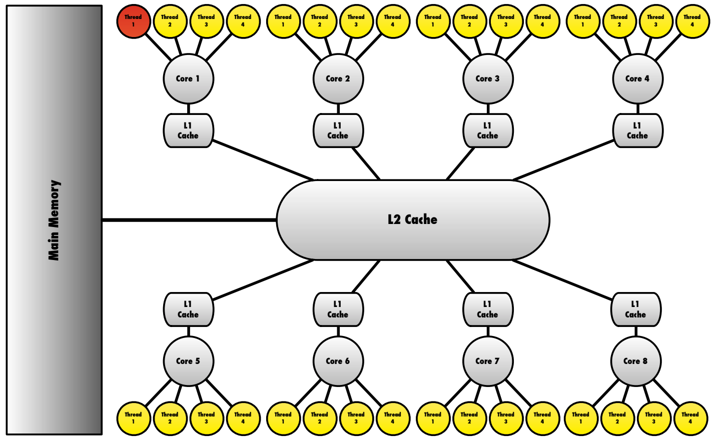
Implementation choices
can impact schedulability as much as
scheduling-theoretic tradeoffs.

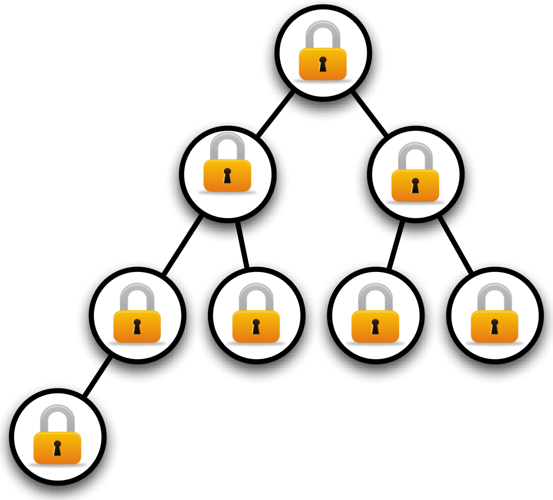
Unless task counts are very high
or periods very short,
G-EDF **can scale** to 32 processors.

Recommendation

<u>Best results obtained with combination of</u>:

fine-grained heap
event-driven scheduling
dedicated interrupt handling





Future Work

Platform.

➡ Repeat study on embedded hardware platform.

Implementation.

➡ Simplify locking requirements.
➡ Parallel mergeable heaps?

Analysis.

➡ Less pessimistic hard real-time G-EDF schedulability tests.
➡ Less pessimistic interrupt accounting.