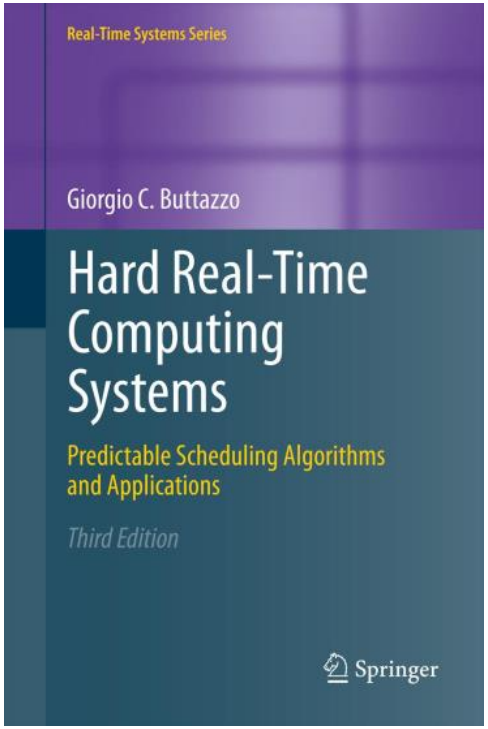
2IMN20 - Real-Time Systems

Non-preemptive Scheduling

Nidhi Srinivasan

2023-2024

Sources
Book: chapter 8



Paper 1:

Mitra Nasri and Gerhard Fohler, "Non-Work-Conserving Non-preemptive Scheduling: Motivations, Challenges, and Potential Solutions," in the Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), 2016, pp. 165-175.

Paper 2:

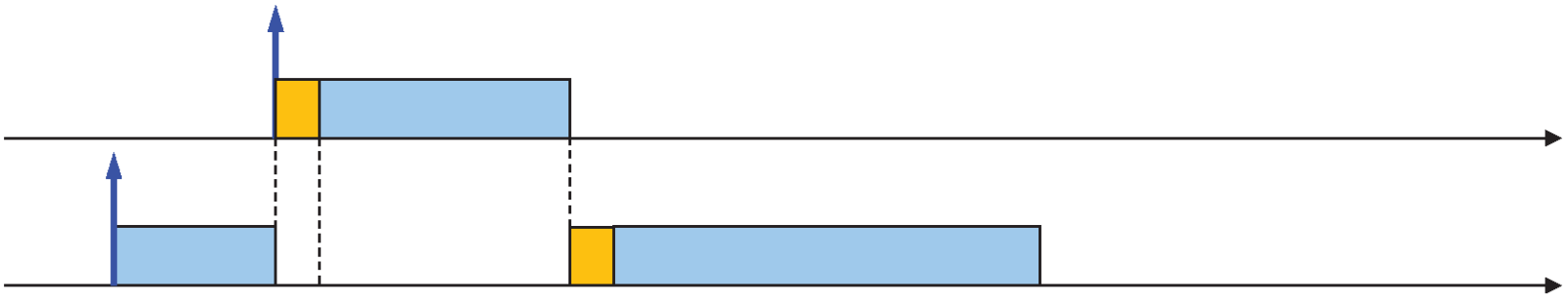
Mitra Nasri and Björn B. Brandenburg, "An Exact and Sustainable Analysis of Non-Preemptive Scheduling", in the Proceedings of the Real-Time Systems Symposium (RTSS), 2017, pp. 1-12

Disclaimer:
Many slides were provided by Dr. Mitra Nasri
Some slides have been taken from Giorgio Buttazzo's course

Disadvantages of preemptions

1- Context switch cost

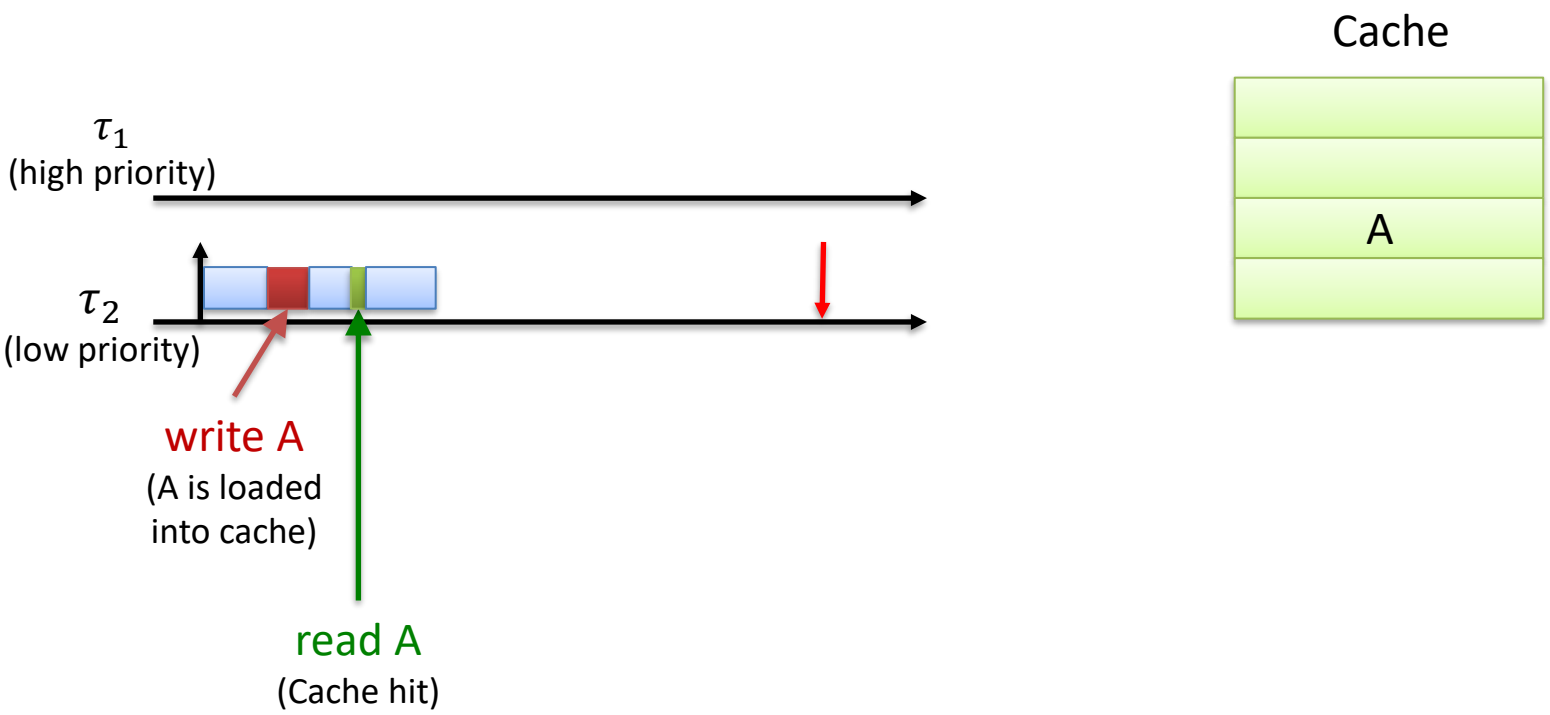
It is the time taken by the scheduler to suspend the running task, switch the context, and dispatch the new incoming task.



Disadvantages of preemptions

2- Cache-related preemption delay (CRPD)

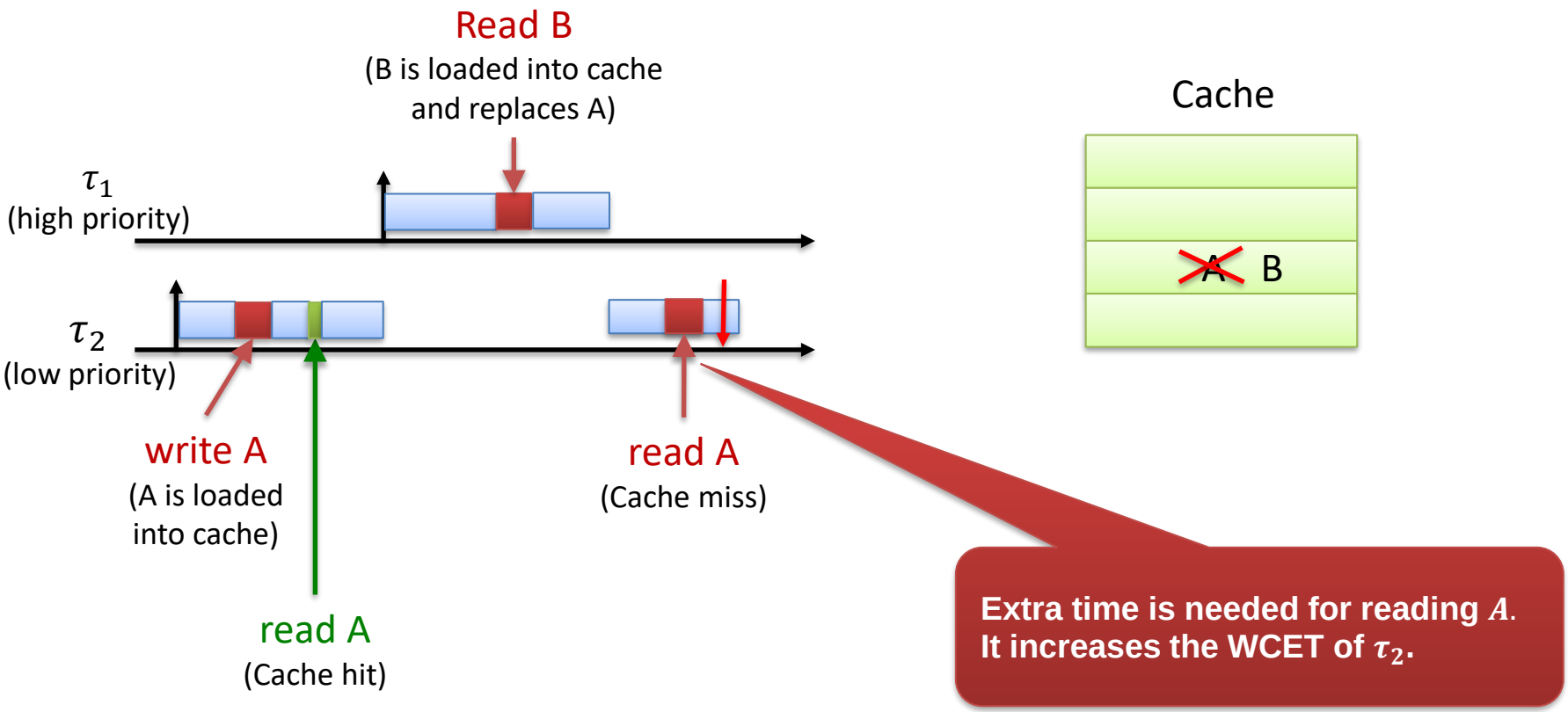
It is the delay introduced by high-priority tasks that evict cache lines containing data used in the future:



Disadvantages of preemptions

2- Cache-related preemption delay (CRPD)

It is the delay introduced by high-priority tasks that evict cache lines containing data used in the future:



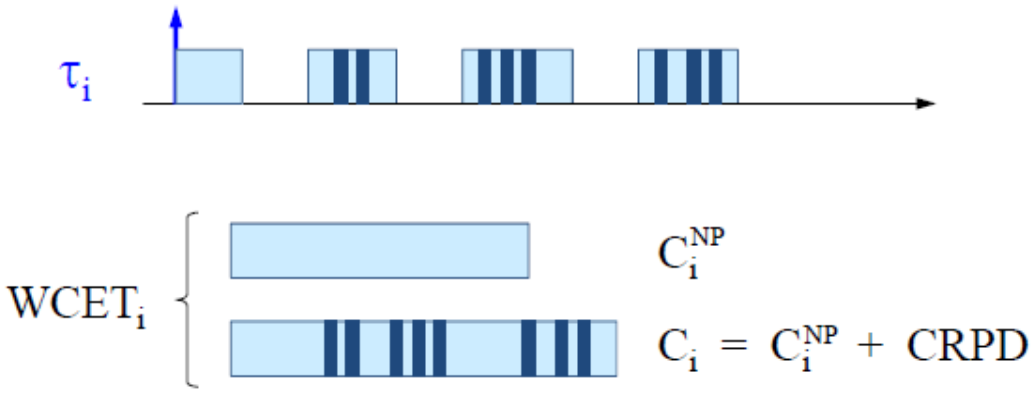
Disadvantages of preemptions

3- larger worst-case execution time

Preemptions cause CRPD which in turn increases the WCET of the task

The amount of CRPD depends on the number of preemptions a job suffers

Task experiencing preemptions by higher priority tasks:



NP stands for non-preemptive

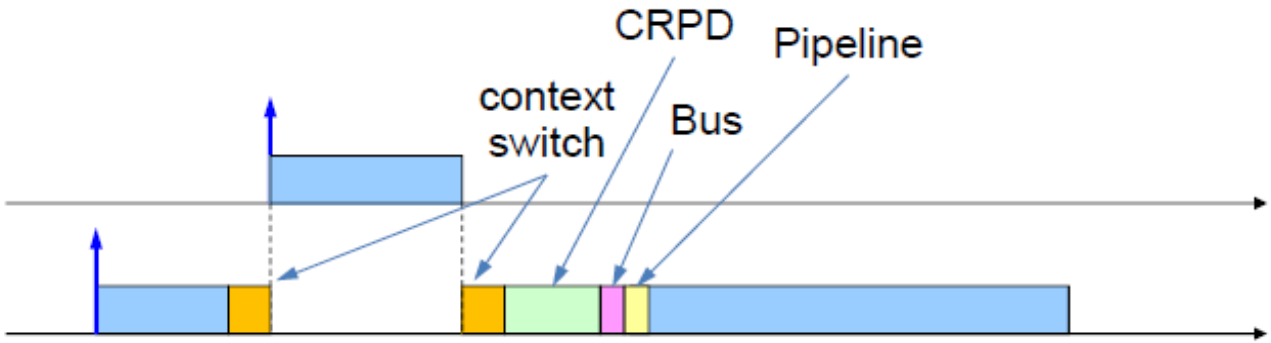
Disadvantages of preemptions

4- Pipeline cost

time to flush the pipeline when a task is interrupted and to refill it when task is resumed.

5- Bus cost

time spent waiting for the bus due to additional conflicts with I/O devices, caused by extra accesses to the RAM for the extra cache misses.

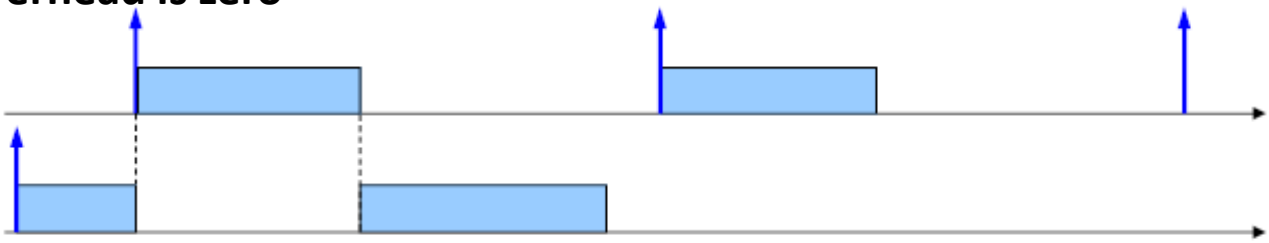


Disadvantages of preemptions

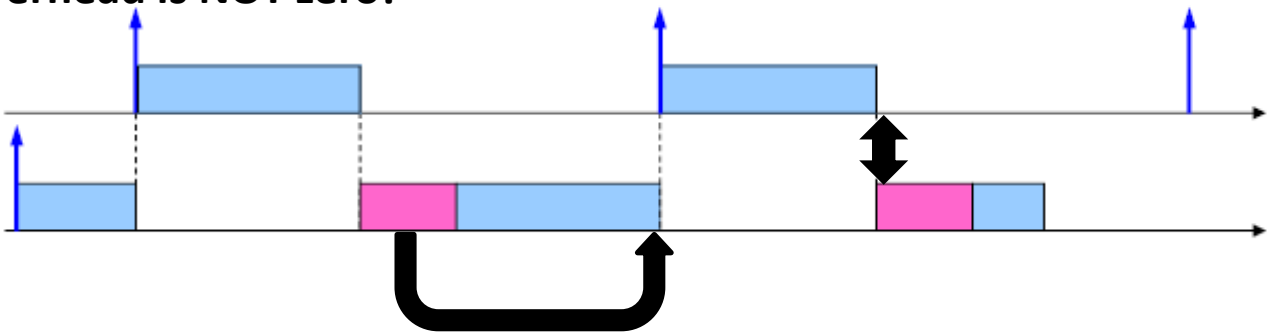
A preemption may cause more preemptions

the extra execution time also increases the number of preemptions:

Preemption overhead is zero



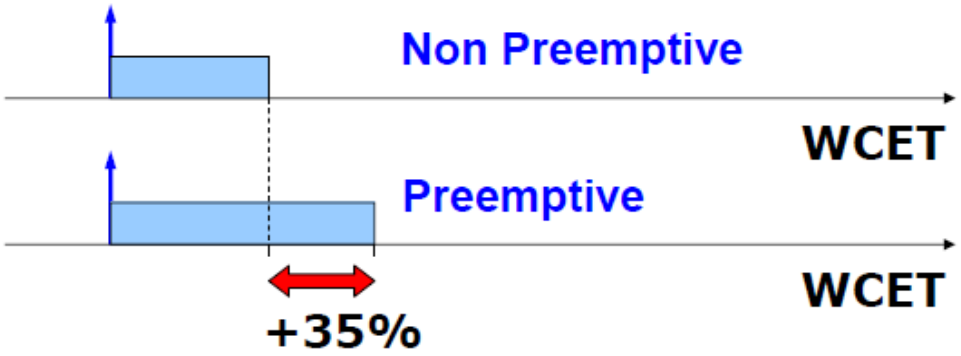
Preemption overhead is NOT zero!

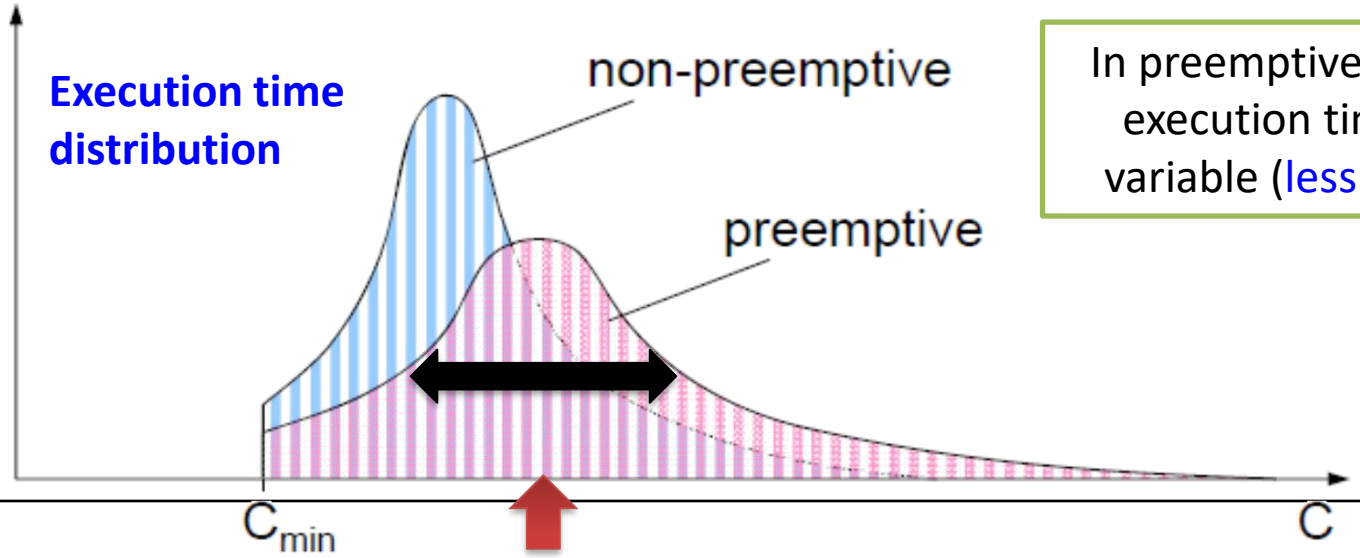


Disadvantages of preemptions

Preemption cost can be very large

- WCETs may increase up to 35% in the presence of preemptions (less efficiency)!





In preemptive execution, the execution times are more variable (less predictability)

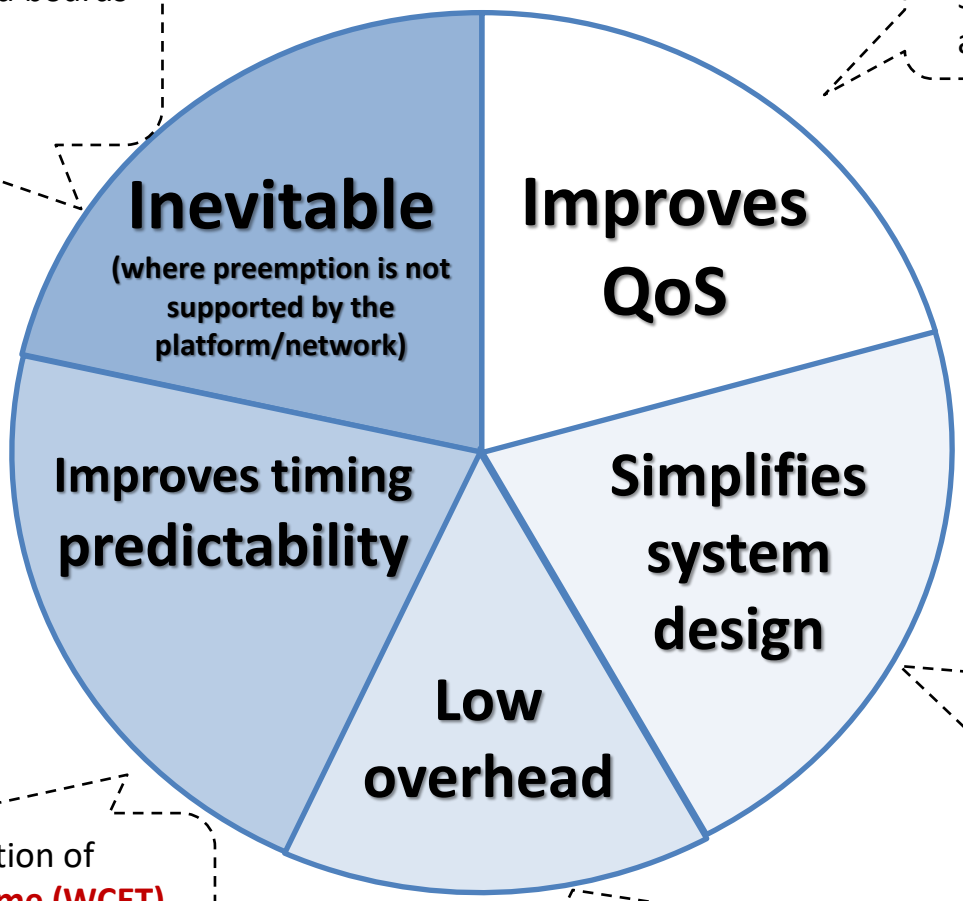
Advantages of non-preemptive scheduling

- It reduces **context-switch overhead**:
  - Making WCETs smaller and more predictable
- It simplifies the **access to shared resources**:
  - No semaphores are needed for critical sections
- It reduces **stack size**:
  - Task can share the same stack, since no more than one task can be in execution
- It allows achieving **small I/O Jitter**:
  - "finishing_time – start_time" has a low variation

Why non-preemptive scheduling?

Examples
- Most bare-metal embedded boards
- GPU device
- Hardware accelerators
- CAN bus

- **Control systems** are sensitive to I/O delay and preemptions



- Simpler resource management policies
- Grants exclusive resource access

- A more accurate estimation of **worst-case execution-time (WCET)**
- More predictable cache

- Reduces context switches
- Avoids intra-task **cache-related preemption delays (CRPD)**

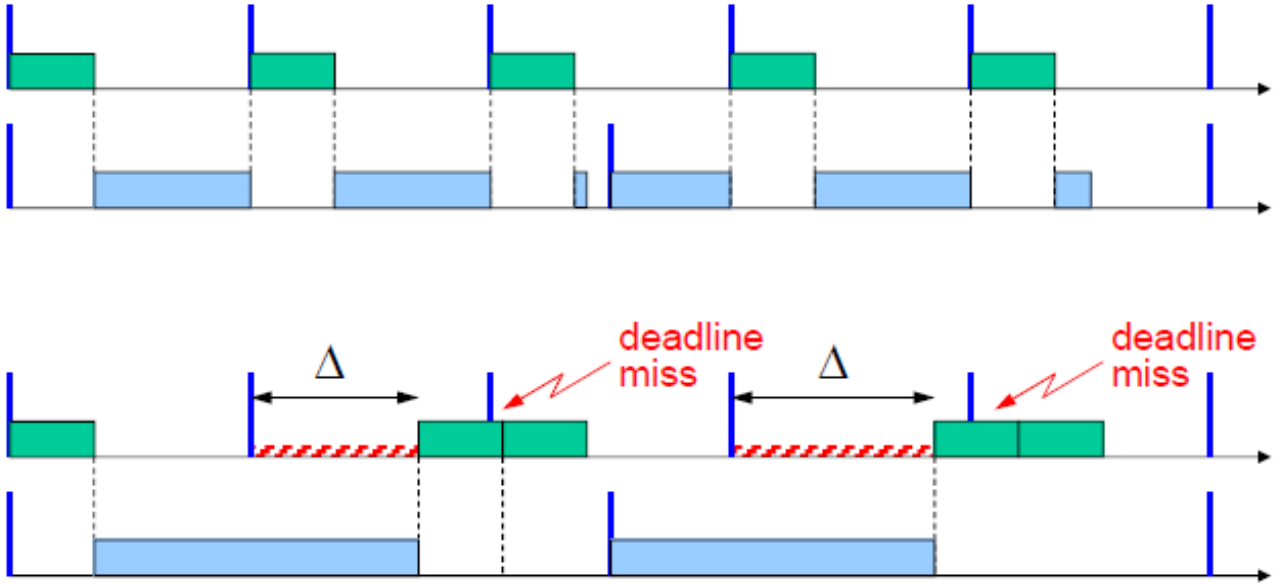
So, why

preemptive

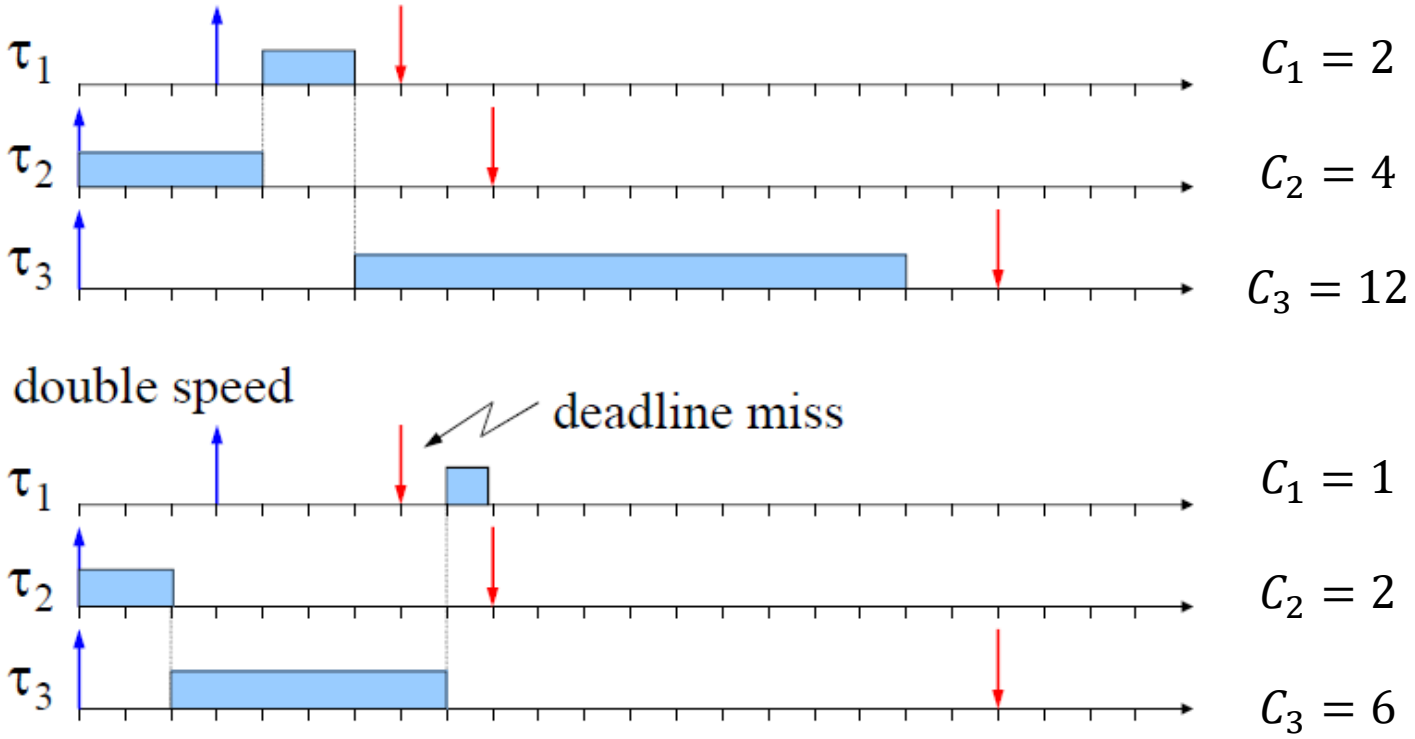
execution?

Disadvantages of non-preemptive scheduling

- In general, non-preemptive scheduling reduces schedulability because of introducing **blocking delays** on the high-priority tasks



Disadvantages of non-preemptive scheduling

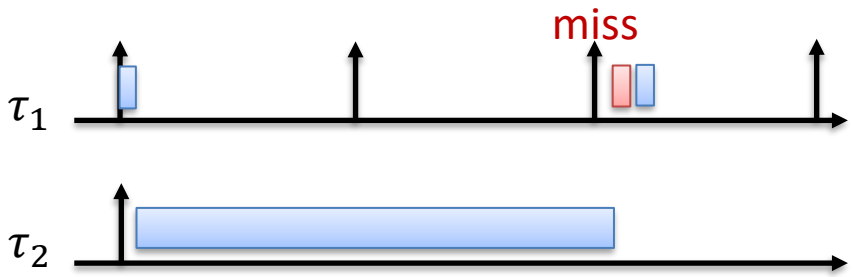


Anomaly is a situation in which a deadline miss happens when we don't expect it to happen!

Disadvantages of non-preemptive scheduling

Build a task set whose utilization is almost 0 and is not schedulable by ANY non-preemptive scheduling algorithm

- The utilization bound under non-preemptive scheduling drops to zero



$C_1 = \epsilon \sim 0 \quad T_1 = 10 \Rightarrow \quad U_1 = \frac{\epsilon}{T_1} \sim 0$

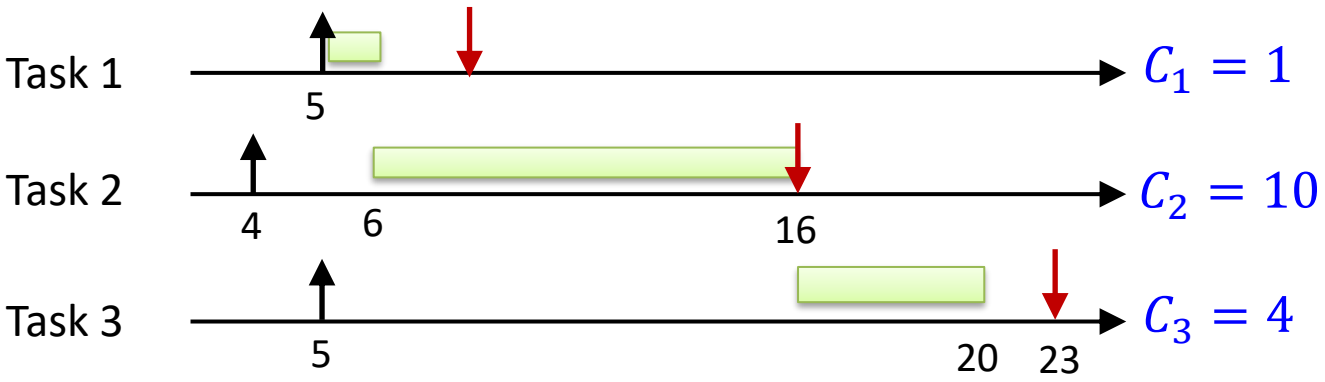
$C_2 = 2T_1 \quad T_2 = \infty \quad \Rightarrow \quad U_2 = \frac{2T_1}{\infty} \sim 0$

$$\boldsymbol{U_1 + U_2 \sim 0}$$

Infeasible! Despite having $U \sim 0$

Yet, for all of its benefits, we look into the non-preemptive scheduling in this lecture

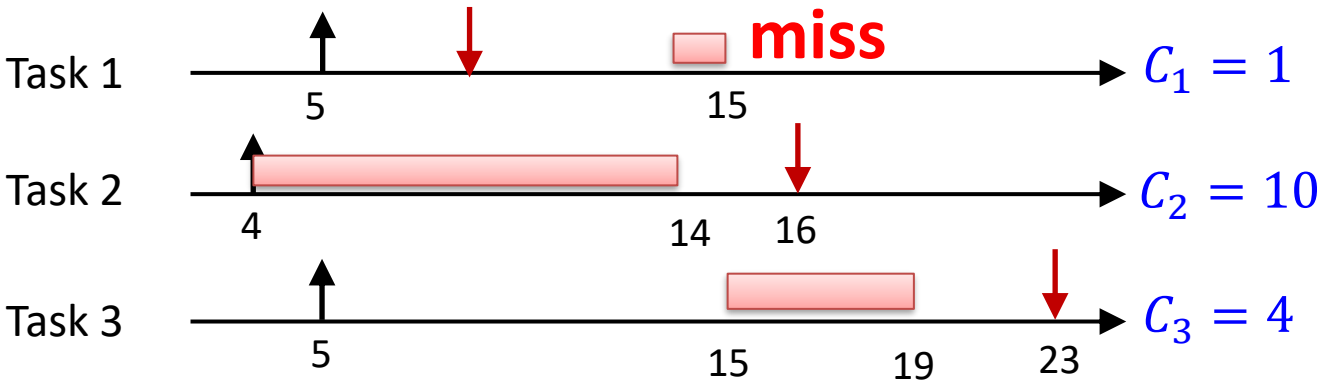
Non-preemptive scheduling



Is this task (job) set feasible?

Yes! It is feasible using a non-work-conserving schedule

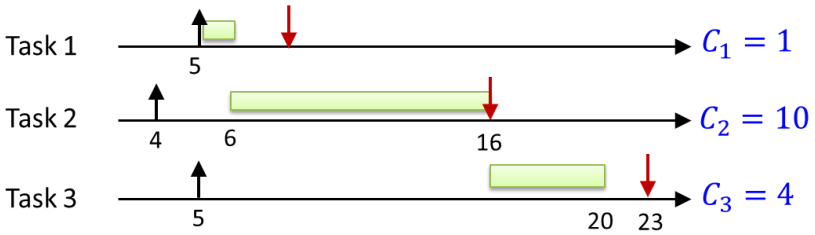
Non-preemptive scheduling



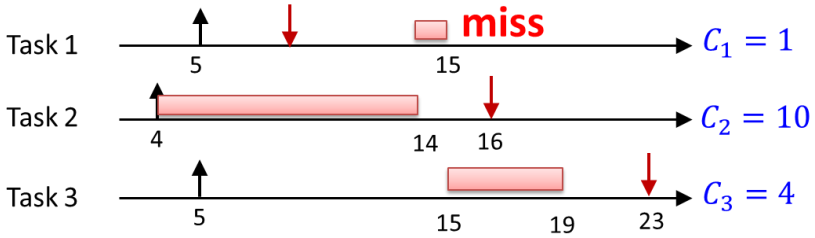
Is this task set schedulable by non-preemptive EDF?

NO

Non-preemptive scheduling



Non-work conserving schedule
Feasible



Non-preemptive EDF
Not Schedulable

The proof comes from Dertouzos 1974:

If a task set with $U \leq 1$ has a feasible schedule σ, then that schedule can be converted to a schedule that respects EDF rules by following Dertouzos transformation.

Does not hold for non-preemptive, work-conserving scheduling policies!

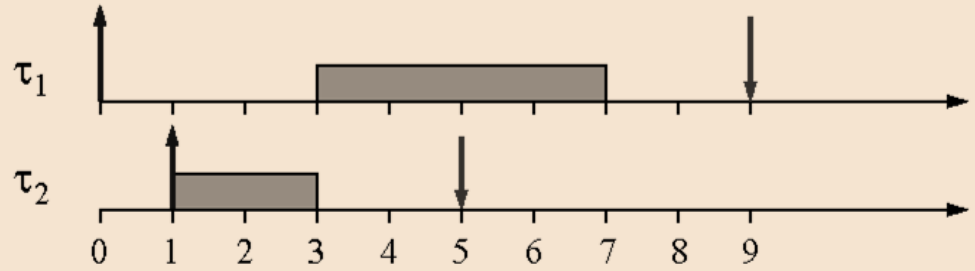
Non-preemptive scheduling

Non-preemptive EDF is not optimal for non-preemptive scheduling

There does not exist any optimal work-conserving scheduling policy for non-preemptive tasks!
(proof is the previous counter example)

Non Preemptive Scheduling

To achieve optimality, an algorithm should be clairvoyant, and decide to leave the CPU idle in the presence of ready tasks:



If we forbid to leave the CPU idle in the presence of ready tasks, then EDF is optimal.

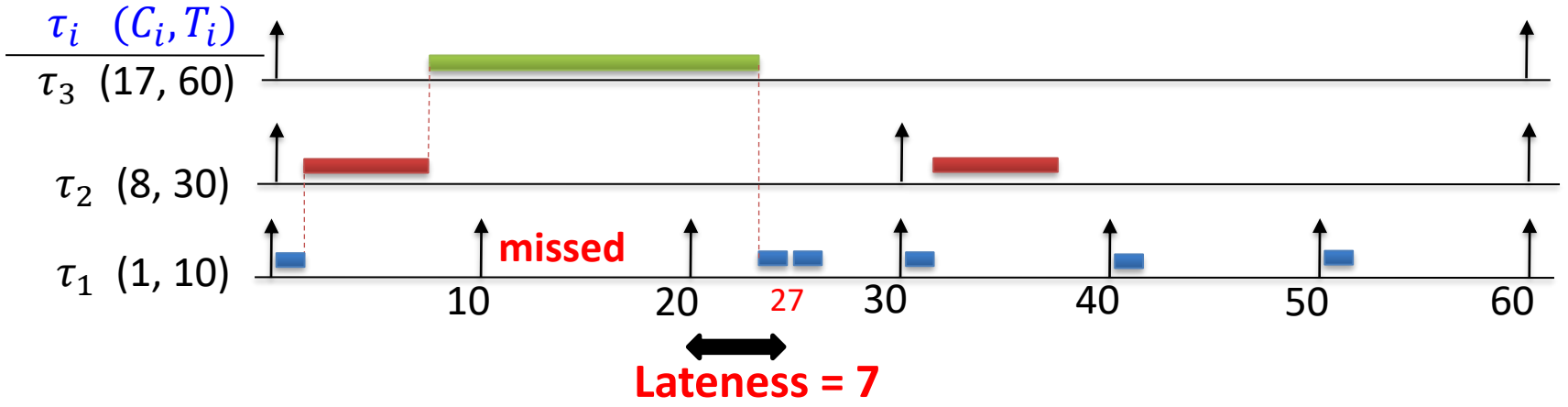
NP-EDF is optimal among
non-idle scheduling algorithms

For a long time, researchers believed so

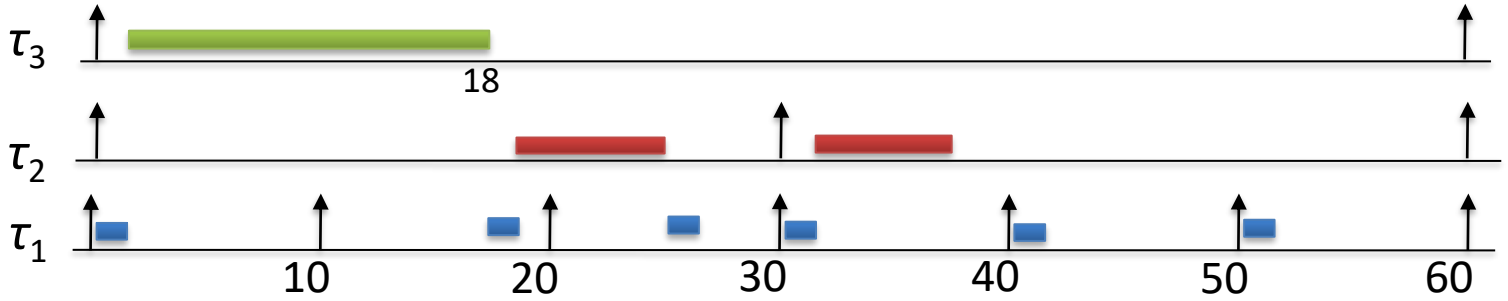
It was only in 2016 that a counter example was found!

EDF is good, but not THAT good!

Non-preemptive EDF (work-conserving)



Non-preemptive fixed priority (work-conserving): $P_1 < P_3 < P_2$



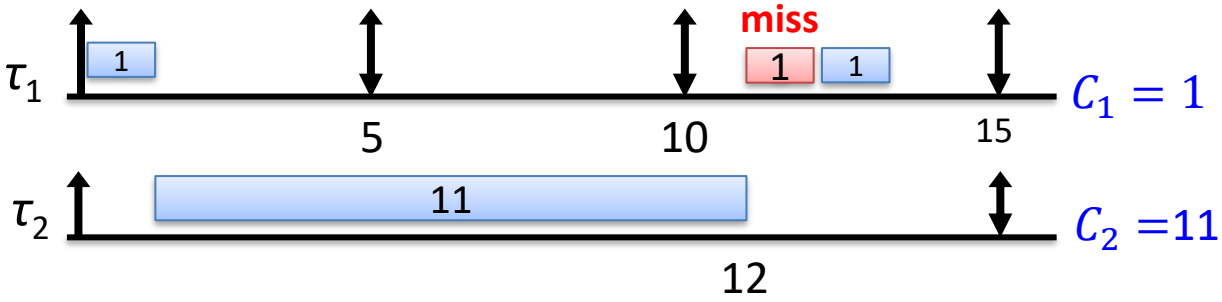
ECRTS'2016 conference:

Mitra Nasri and Gerhard Fohler, "Non-Work-Conserving Non-preemptive Scheduling: Motivations, Challenges, and Potential Solutions," in the Euromicro Conference on Real-Time Systems (ECRTS), 2016, pp. 165-175.

Take-away message

- EDF is a good policy for **preemptive** tasks but not necessarily for non-preemptive tasks!
- It is **not an optimal** policy among all non-preemptive work-conserving scheduling policies

A necessary test for non-preemptive scheduling



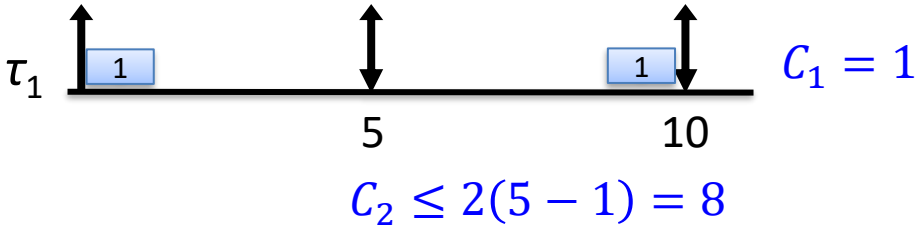
Is this task set feasible?

No! Whatever you do, it is impossible to find a feasible schedule for this task set

Let's design a necessary test together

$U \leq 1$

$\forall i, 1 < i \leq n,\ C_i \leq 2(T_1 - C_1)$

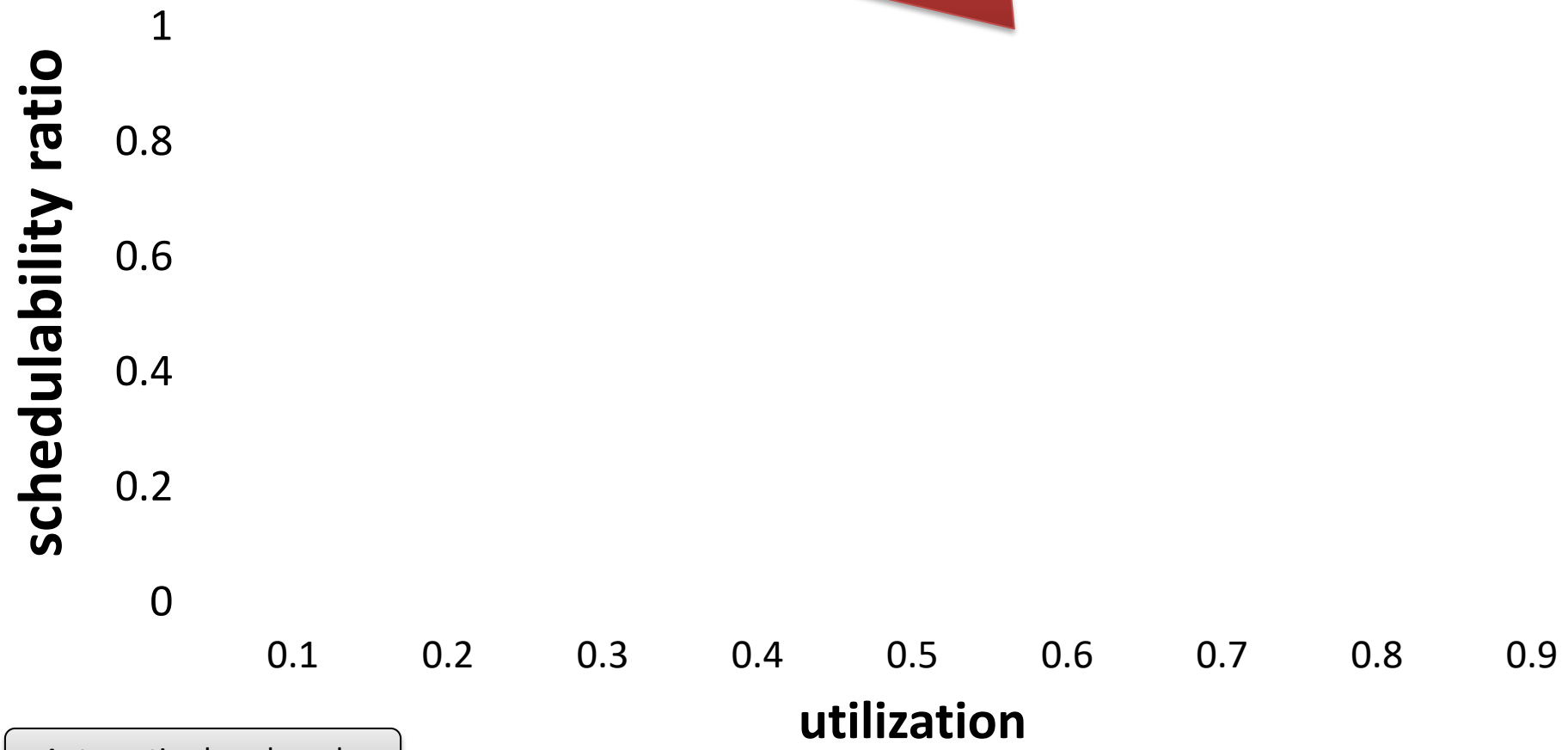


Can we improve it?
Maybe! No body knows!

Assuming that tasks are sorted by their period

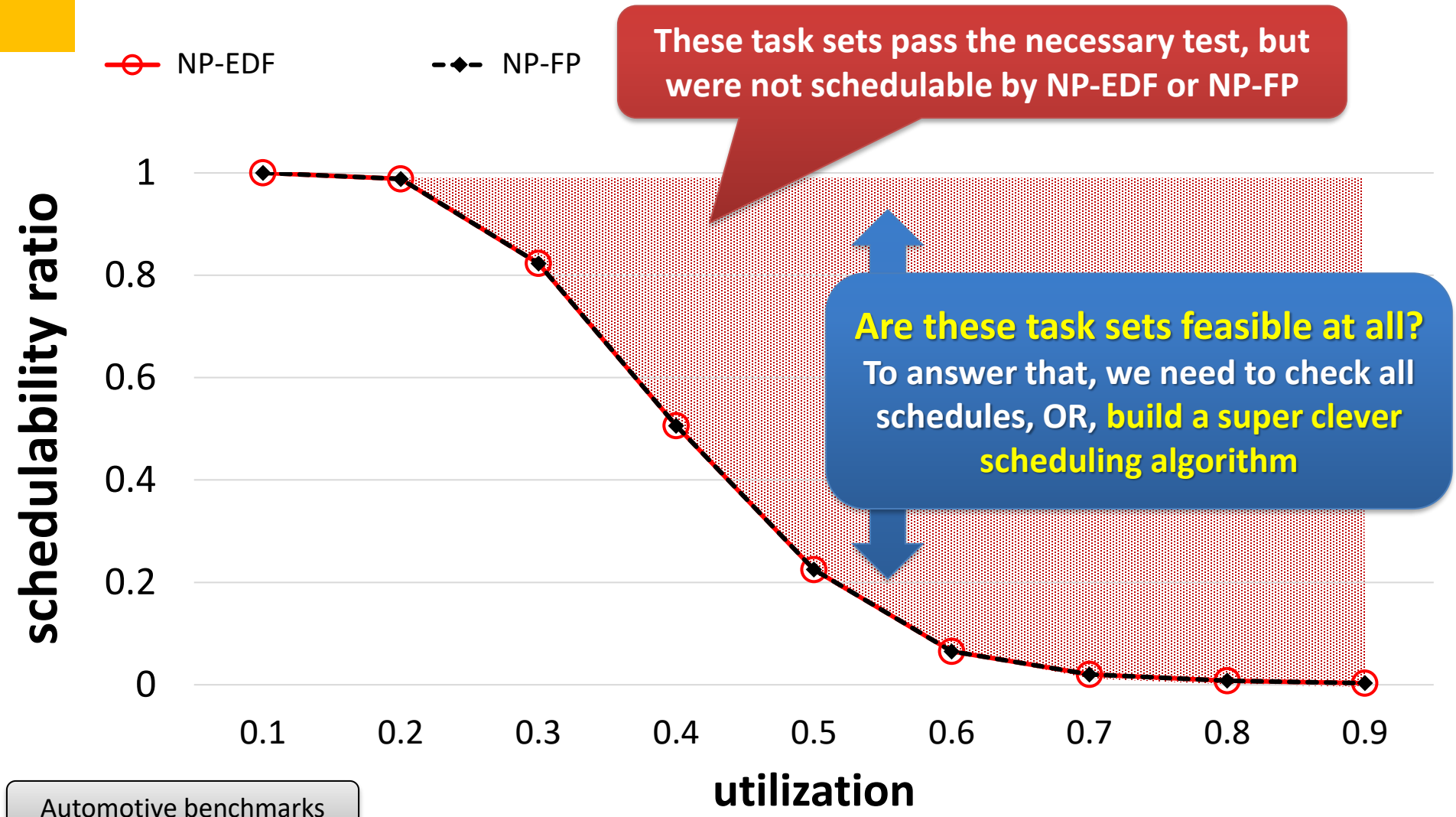
How effective are the non-preemptive EDF (NP-EDF) and non-preemptive fixed-priority (NP-FP) scheduling policies?

We only consider task sets that pass the necessary test



Automotive benchmarks

Sufficient v.s. exact tests for NP-FP (and NP-EDF)

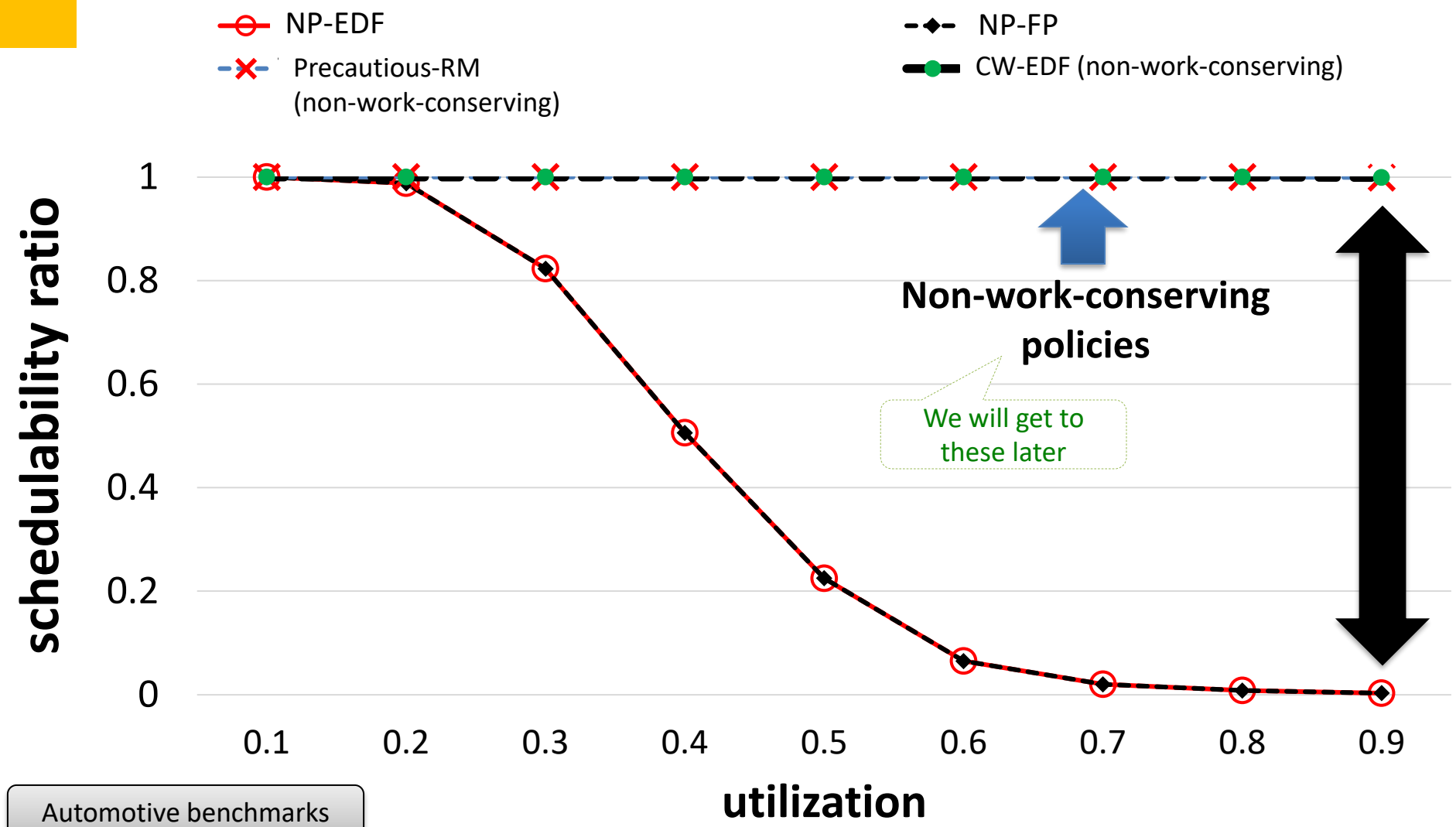


Automotive benchmarks

We found the “**clever**” trick!

Online non-work-conserving policies

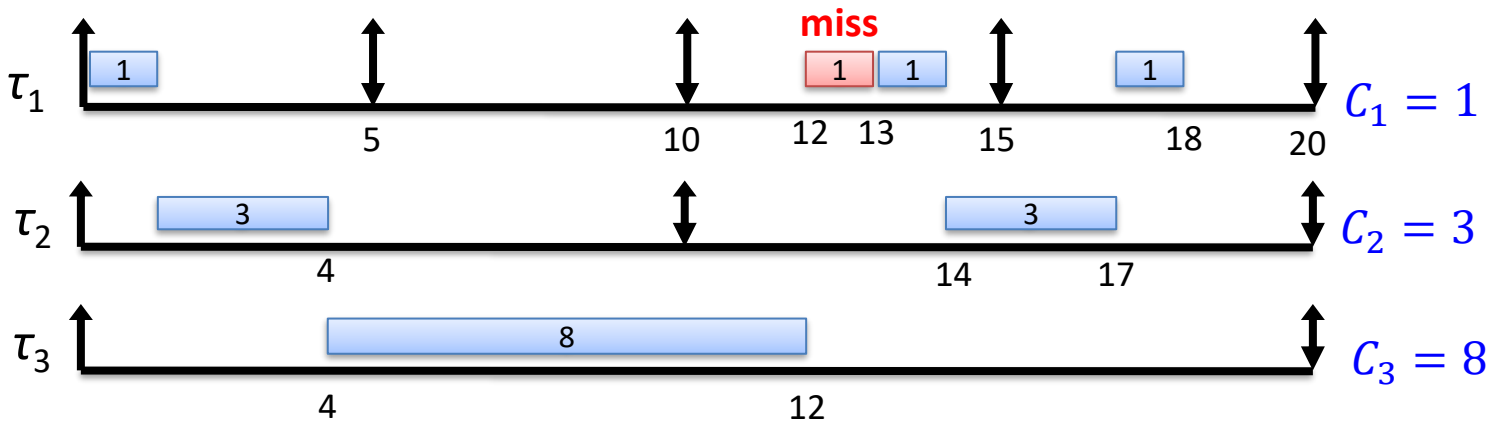
Non-work-conserving scheduling!



Automotive benchmarks

Why non-preemptive scheduling is hard?

NP-FP (with rate-monotonic priorities) schedule:



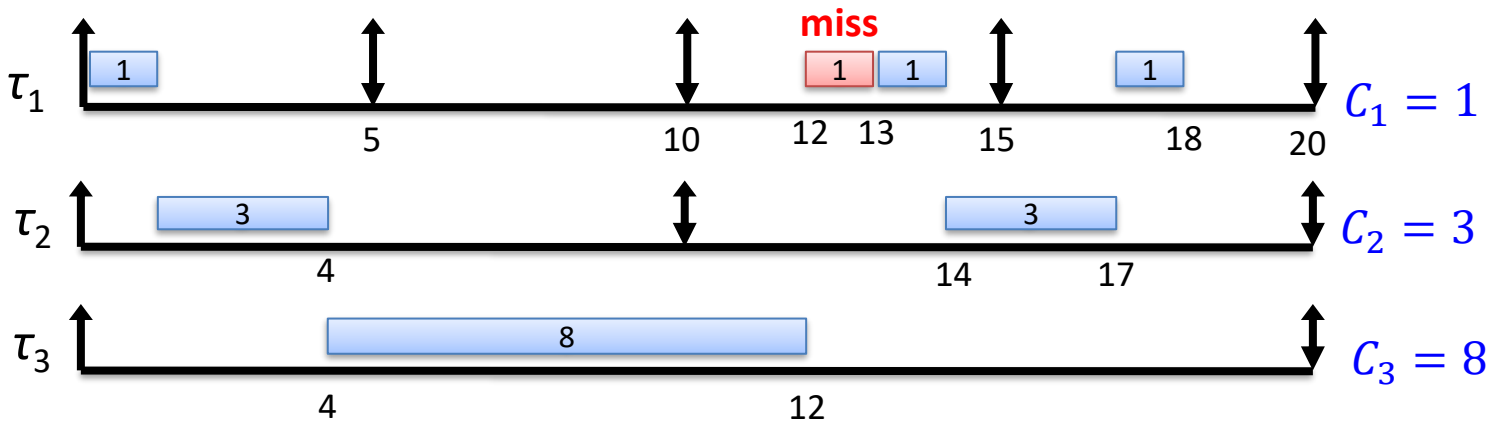
Why there is a deadline miss?

Because at time 4, NP-FP took a **wrong** decision!

This wrong decision resulted in a loooong blocking time on $J_{1,2}$

Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:

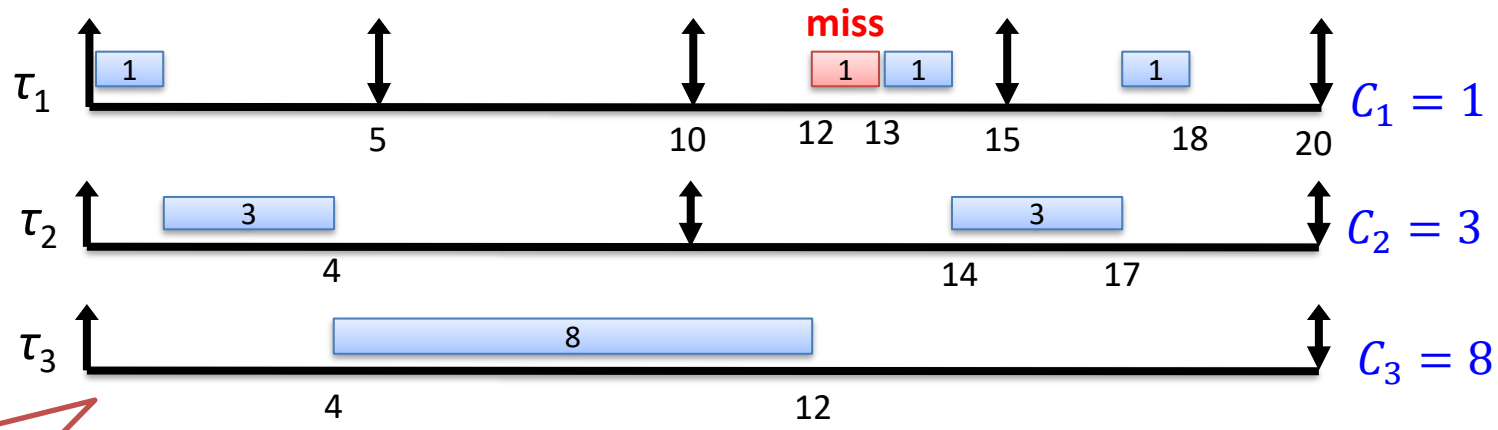


Which task do you think will likely have a lot of deadline misses in the future?

τ_1 because it has the smallest period (highest frequency of activation)

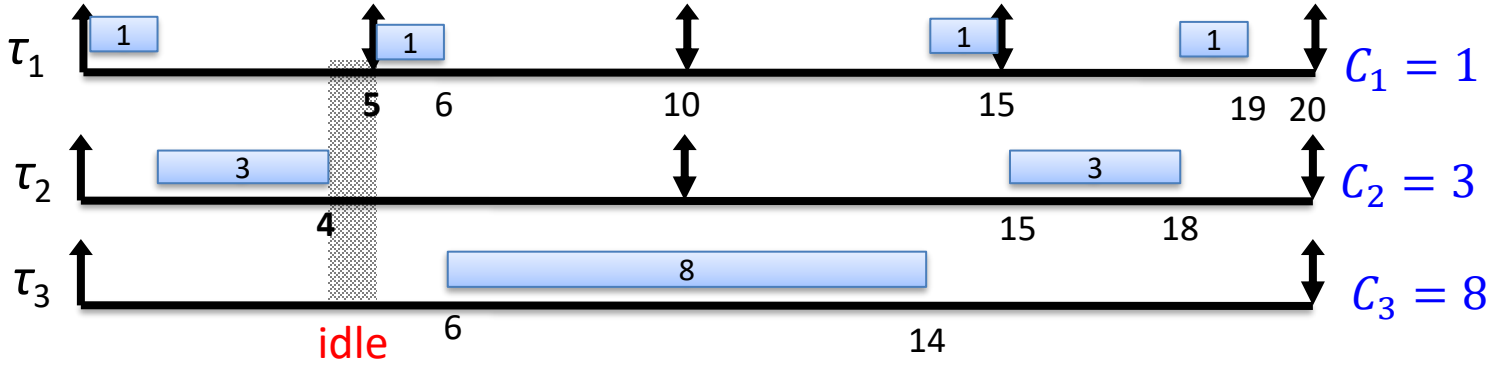
Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:



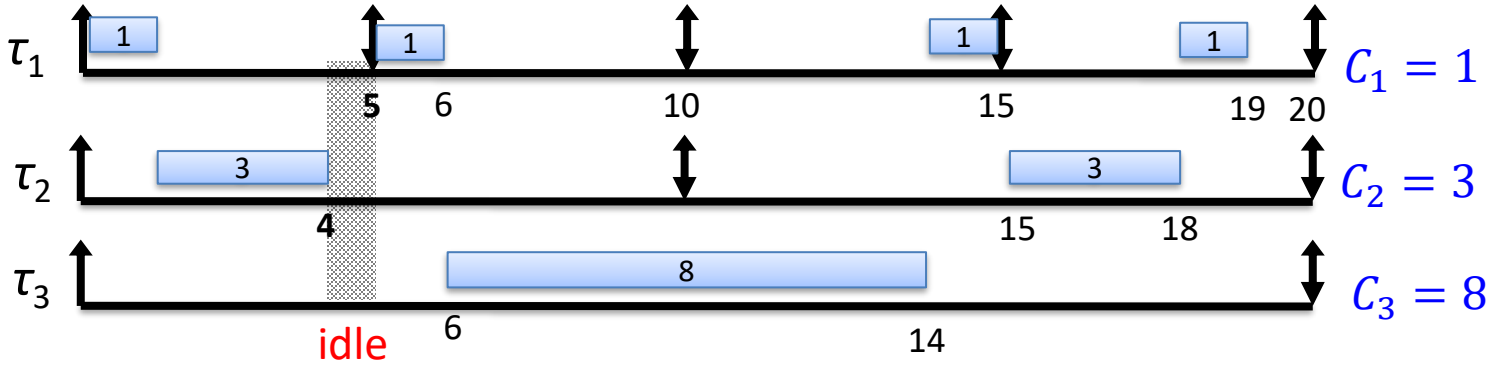
How could this situation be avoided?

By <u>not</u> scheduling $J_{3,1}$ at time 4



Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:

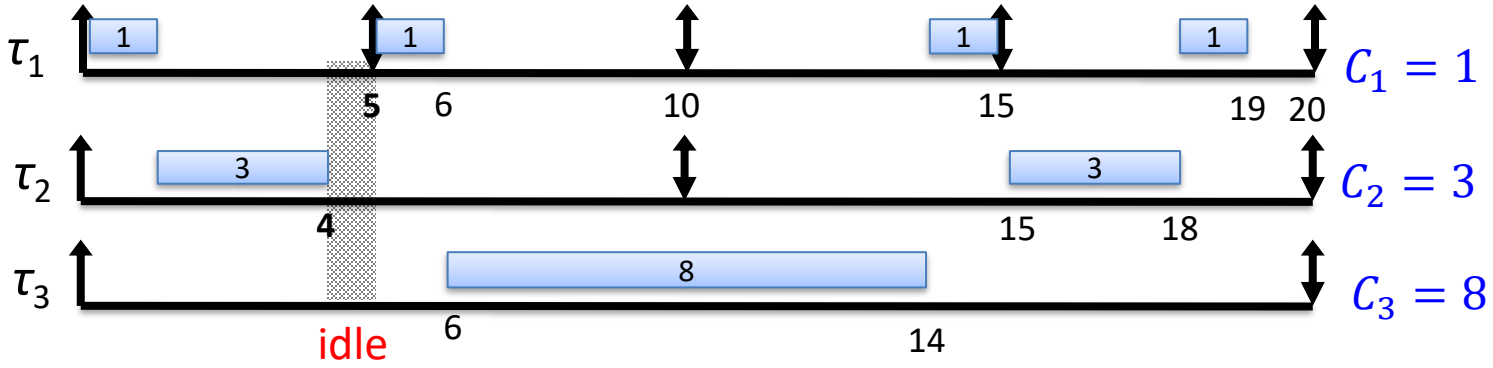


How should the scheduler know that it MUST leave the processor "idle" instead of executing a task? (try to make a general rule)

By using the "knowledge" about the periodicity of the tasks to predict the immediate future

Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:



How does the periodicity help?

Knowing tasks are periodic, we exactly know when the "future jobs" of the tasks will arrive.

Hence, we can check if our current decision will impact the deadline of those FUTURE jobs!

A general non-work-conserving solution

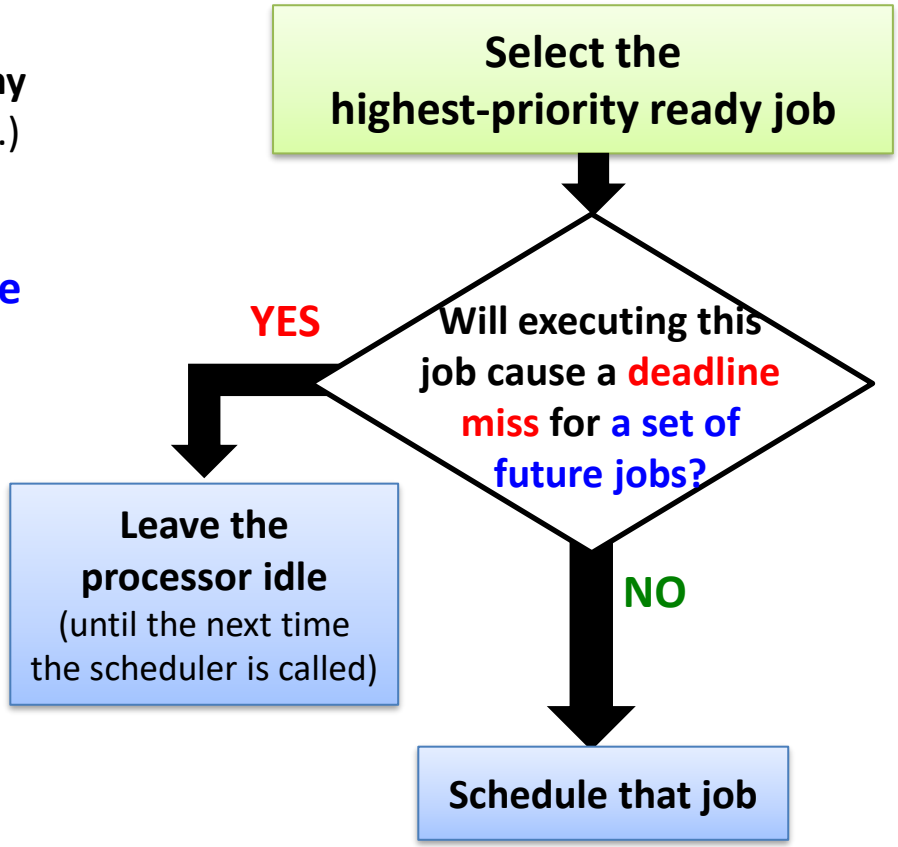
1- Select the highest-priority ready job

(the **highest-priority job** can be chosen by **any scheduling policy** of your choice: EDF, FP, etc.)

2- Check if executing that job will cause a deadline miss for a set of jobs that will arrive in the future

- **Yes?** Then **don't schedule** the current task, instead, **leave the processor idle** until the next job arrives to the system
- **No?** Then schedule the current high-priority ready task

Why just a "set of future jobs"?
Why not for "all future jobs"?

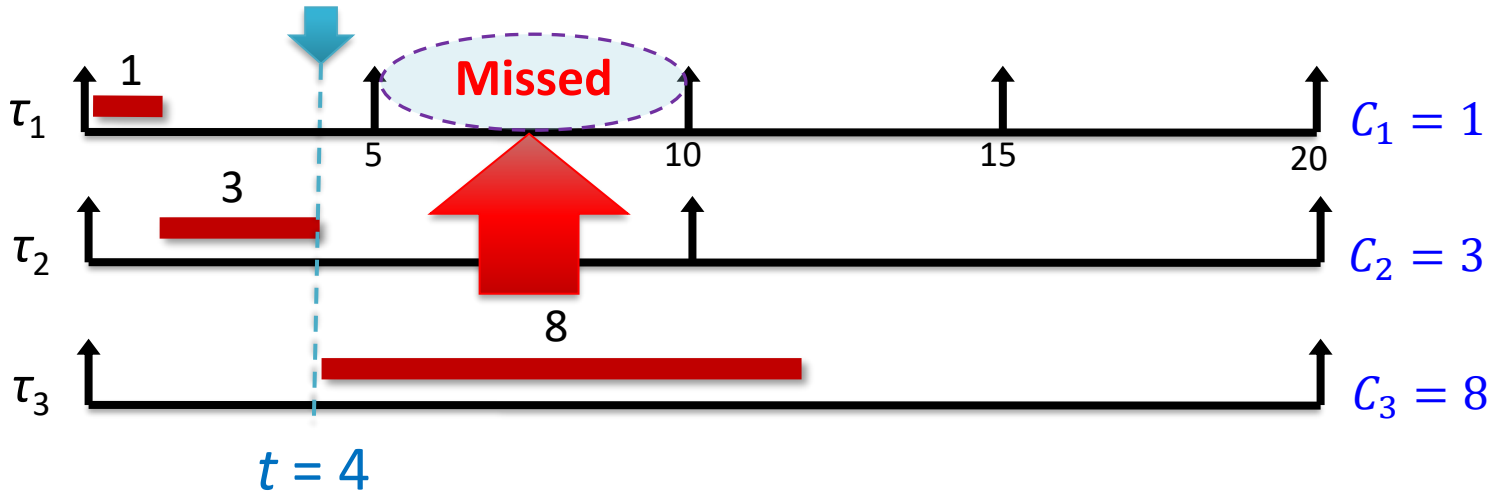


Example: Precautious-RM policy

O(1)

(just look at "one" future job)

- **Select** the highest-priority ready task (using RM priorities)
- **If** (it will cause a deadline miss for the next job of τ_1) **then**
  leave the processor idle
- **Else**
  Dispatch the task

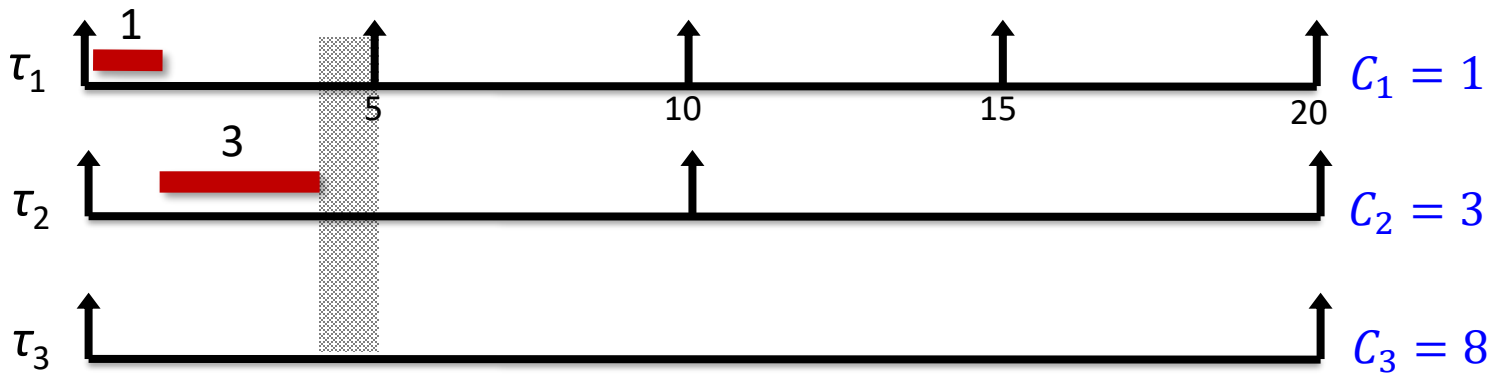


Example: Precautious-RM policy

O(1)

(just look at "one" future job)

- **Select** the highest-priority ready task (using RM priorities)
- **If** (it will cause a deadline miss for the next job of τ_1) **then**
  leave the processor idle
- **Else**
  Dispatch the task

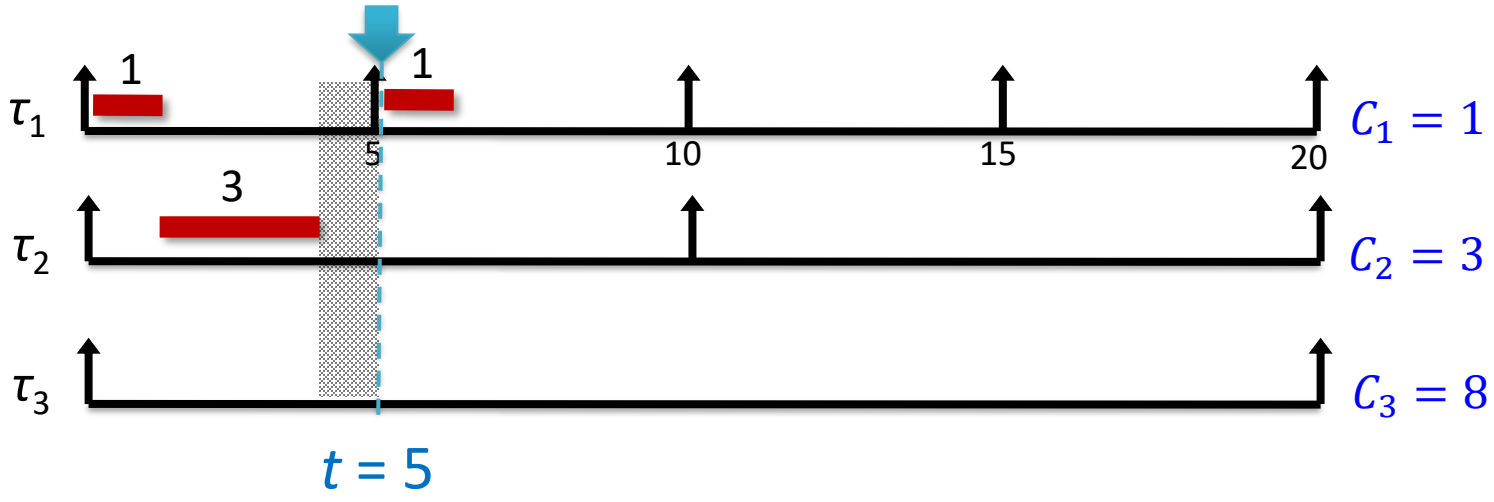


Example: Precautious-RM policy

O(1)

(just look at "one" future job)

- **Select** the highest-priority ready task (using RM priorities)
- **If** (it will cause a deadline miss for the next job of τ_1) **then**
  leave the processor idle
- **Else**
  Dispatch the task

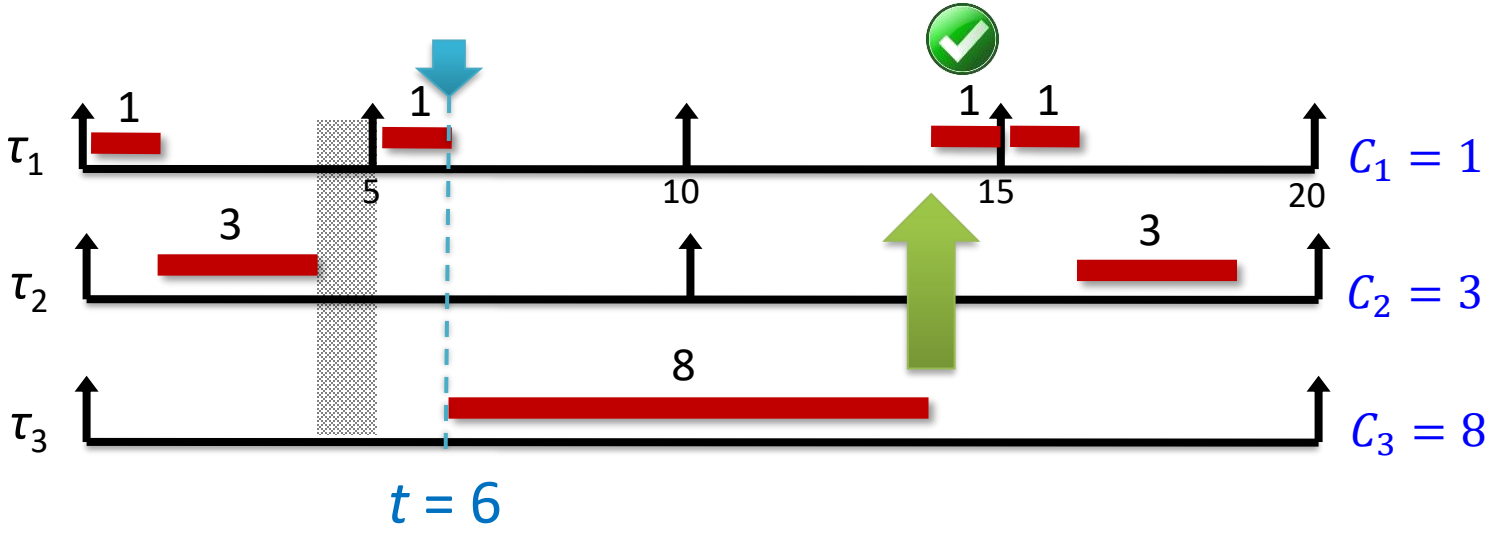


Example: Precautious-RM policy

O(1)

(just look at "one" future job)

- **Select** the highest-priority ready task (using RM priorities)
- **If** (it will cause a deadline miss for the next job of τ_1) **then**
  leave the processor idle
- **Else**
  Dispatch the task



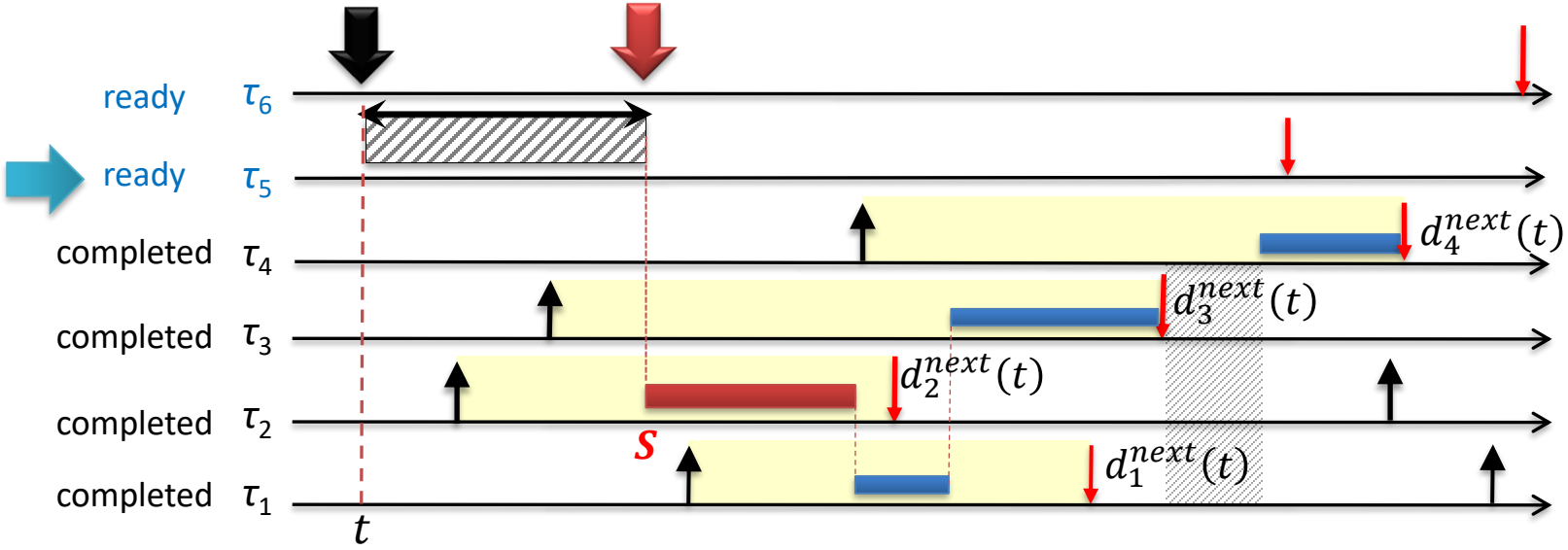
Open problem: how can we improve schedulability while keeping the solution O(1)?

Open problem: how can we design a non-work-conserving algorithm for a multiprocessor system?

Critical-Window EDF (CW-EDF)

O(n)

- Considers the next job of **each task**
- Sorts their deadlines
- Obtains the latest start time of each job starting from the one with the latest deadline
- Checks if the current high priority job can finish before the latest start time ***S***



$d_i^{next}(t)$ = the absolute deadline of the next job of Task i which will be released *after* time t.

[ECRTS 2016]

Important design factors for a scheduling algorithm

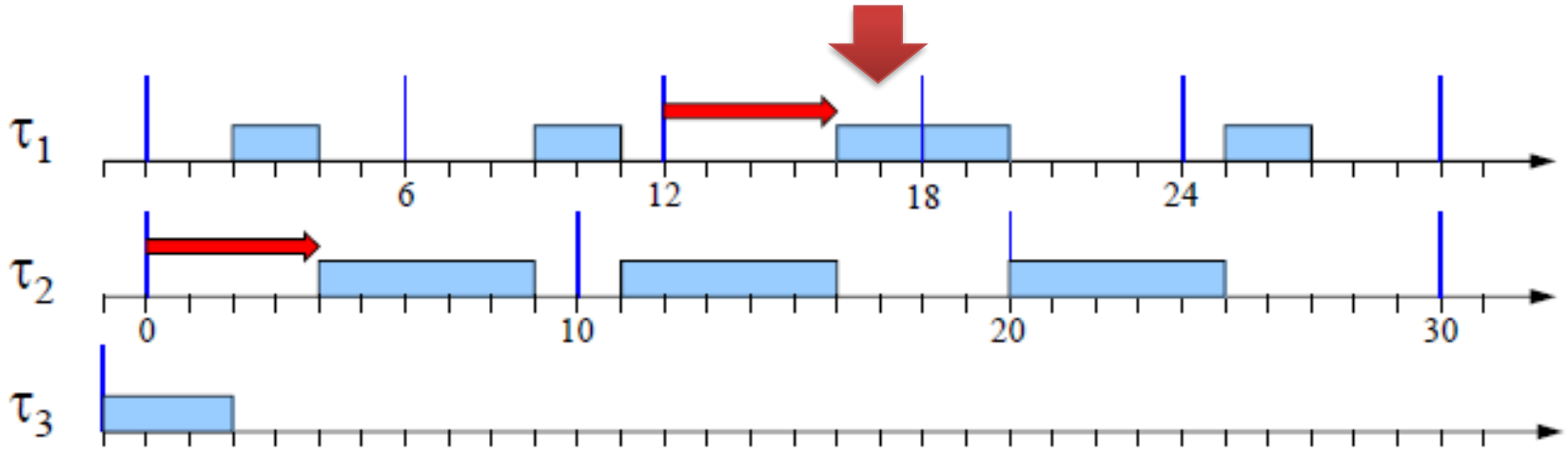
- **Effectiveness**
  - How successful is the algorithm in generating feasible schedules
- **Analyzability**
  - Can you analyze it at all? (can you come up with a schedulability test?)
- **Runtime**
- **Memory requirement**

These are **sufficient** schedulability tests for periodic tasks

Existing schedulability analysis for NP-FP and NP-EDF

Challenges of analyzing non-preemptive systems

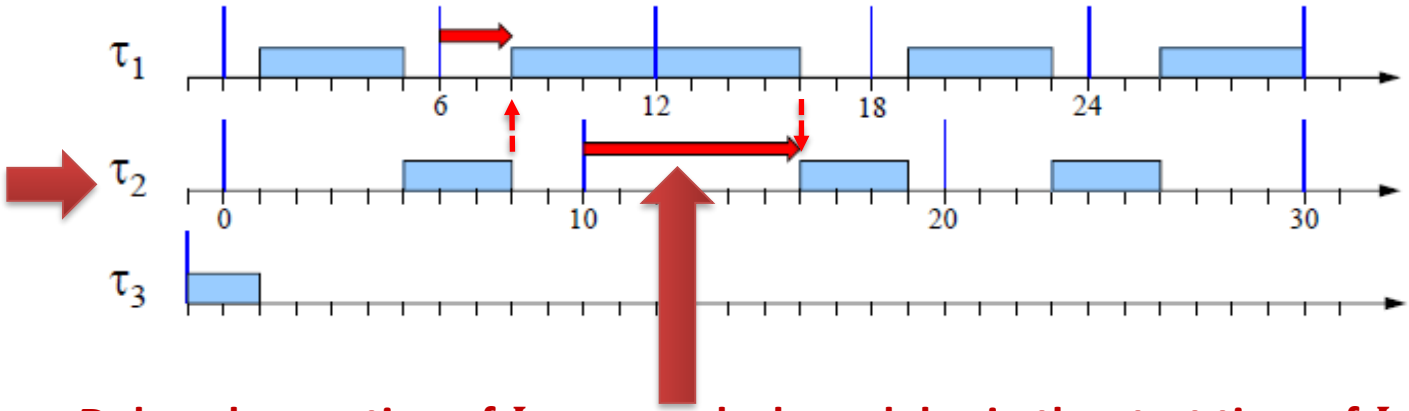
- Analysis of non-preemptive systems is more complex, because the **largest-response time** may **not occur in the first job** after the critical instant.



Challenges of analyzing non-preemptive systems

Self-pushing phenomenon

- High-priority jobs activated during non-preemptive execution of lower-priority tasks are pushed ahead and introduce higher delays in subsequent jobs of the same task.

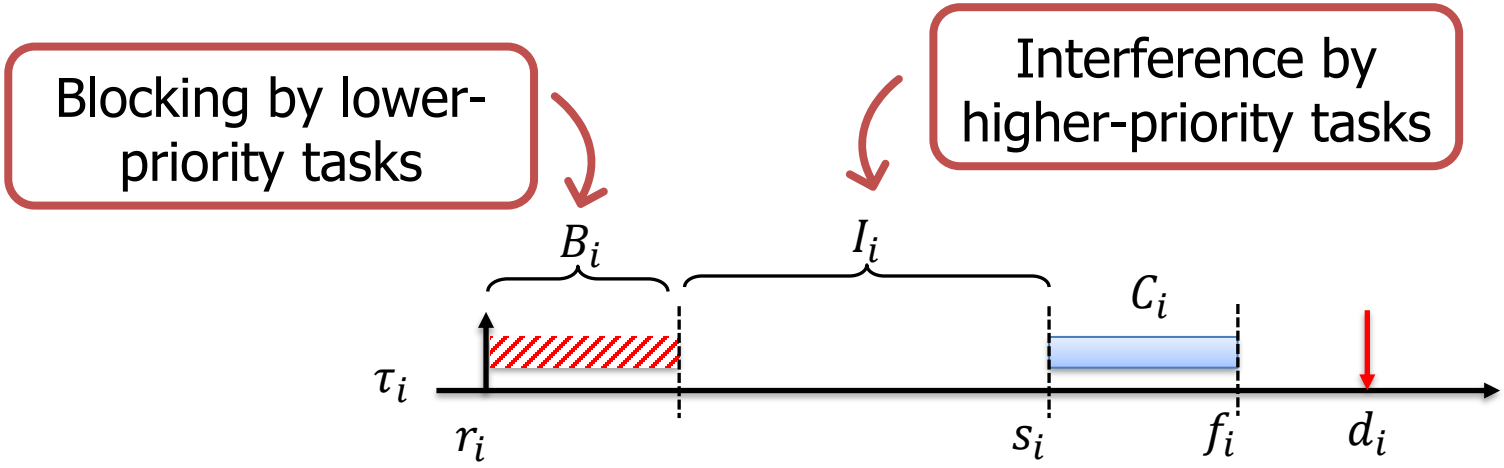


Delayed execution of $J_{2,1}$ caused a long delay in the start time of $J_{2,2}$

Challenges of analyzing non-preemptive systems

- Hence, the analysis of τ_i must be carried out for **multiple jobs**, until all tasks with higher priority than P_i are completed.

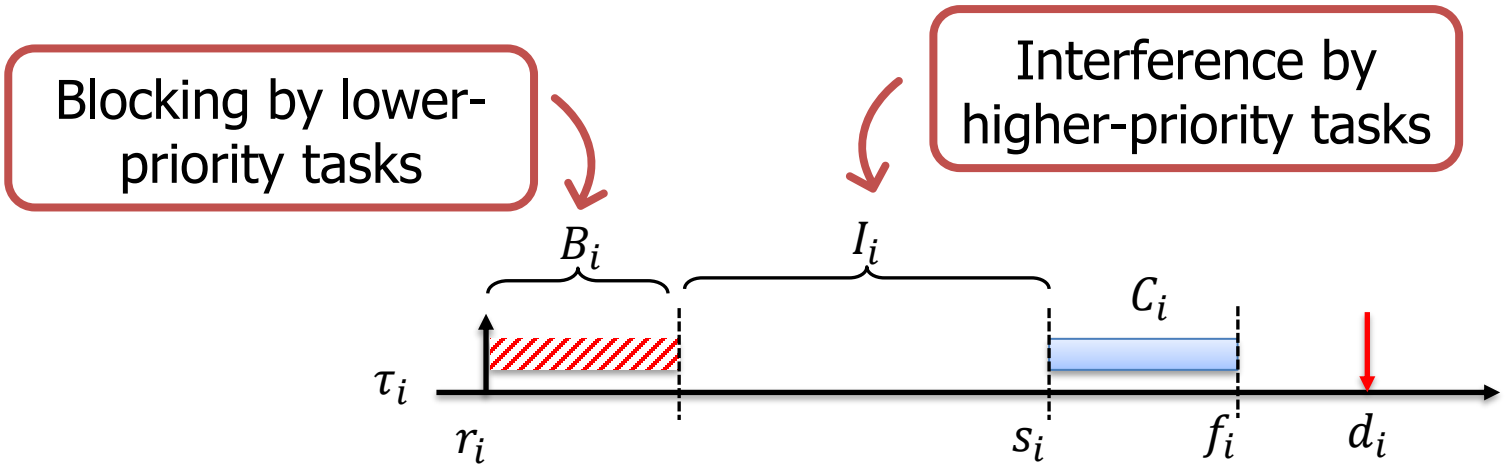
Response-time analysis of NP-FP



Maximum blocking (caused by the **lower-priority tasks**):

$$B_i = \max\{C_j - 1 \mid \forall \tau_j, P_i < P_j\}$$

Response-time analysis of NP-FP



Maximum interference (caused by the **higher-priority tasks**):

I_i : harder to calculate

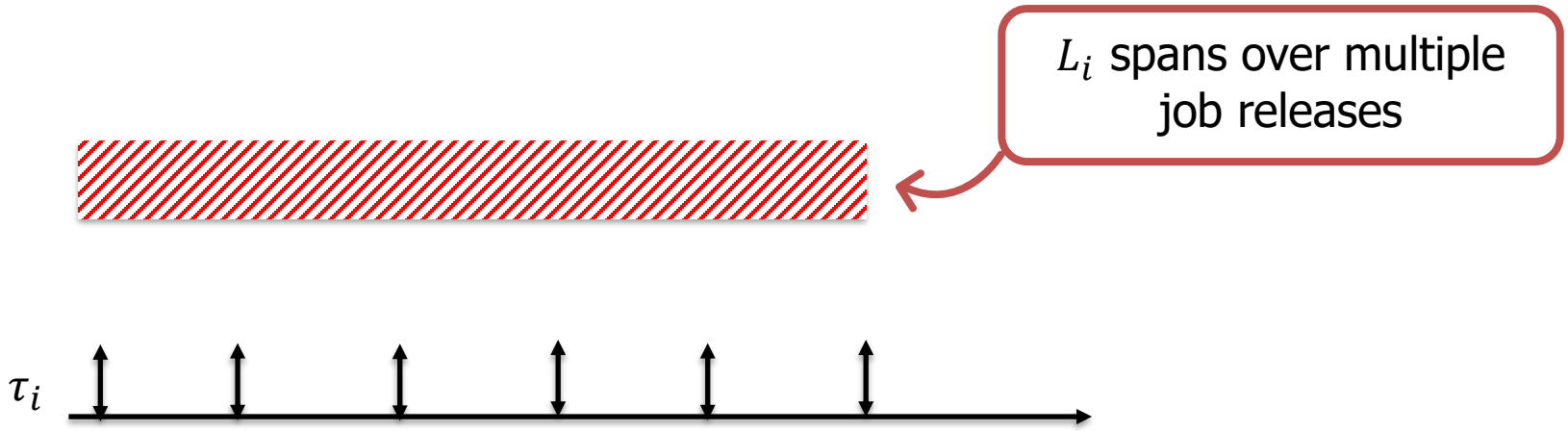
Level-i Active Period:

$$L_i^{(0)} = B_i + C_i$$

$$L_i^{(s)} = B_i + \sum_{P_i<P_j} \left\lceil \frac{L_i^{(s-1)}}{T_j} \right\rceil C_j$$

$$L_i = L_i^{(s)} \text{ when } L_i^{(s)} = L_i^{(s-1)}$$

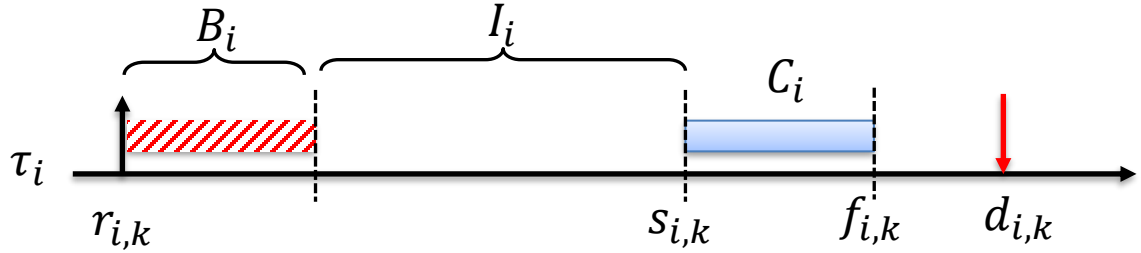
Response-time analysis of NP-FP



Number of jobs to be analysed:

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil$$

Response-time analysis of NP-FP



$$s_{i,k}^{(l)} = B_i + (k-1)C_i + \sum_{P_i<P_j} \left(\left\lfloor \frac{s_i^{(l-1)}}{T_j} \right\rfloor + 1 \right) \cdot C_j$$

NOTE: the end of I_i cannot coincide with the activation of a higher-priority task, because it would increase I_i.
Hence: instead of $\lceil x \rceil$ we must use $\lfloor x \rfloor + 1$

The solution is based on fixed-point iterations:

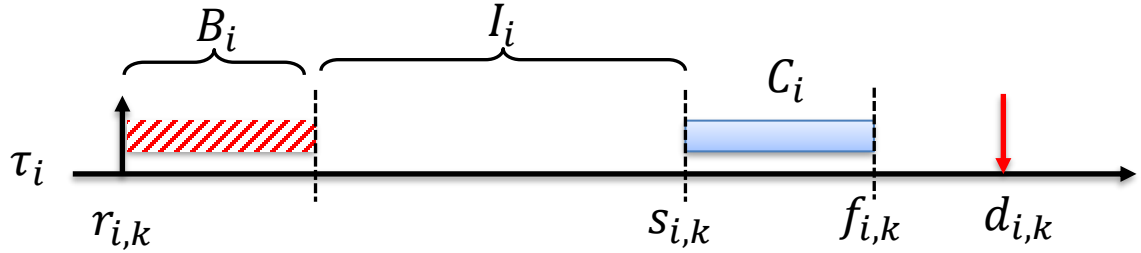
Starting point:

$$s_{i,k}^{(0)} = B_i + \sum_{P_i<P_j} C_j$$

Iterate until:

$$s_{i,k}^{(l)} = s_{i,k}^{(l-1)}$$

Response-time analysis of NP-FP



Finish time:

$$f_{i,k} = s_{i,k} + C_i$$

Response Time:

$$R_i = \max_{k \in [1, K_i]} \{f_{i,k} - (k-1)T_i\}$$

History of the schedulability analysis of NP-FP?

In 1994, Tindel et al. have introduced the test we saw earlier.
They did not know that their test is correct only if the following condition holds:

"task set must be "preemptively feasible" with $D \leq T$"

In the late 90's, this test was used in CAN controllers in the cars for any type of task set!

Only in 2007, Dr. Reinder Bril from TU/e noticed a bug in the test!

However, thankfully, no car accident happened because of this bug.

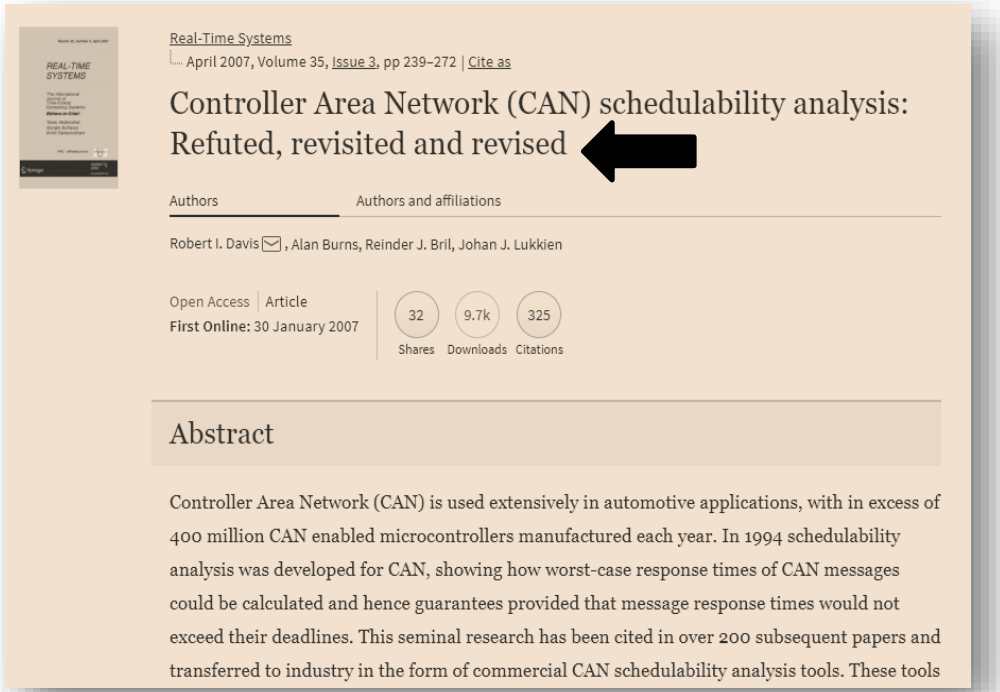
Why?

Because engineers decided to "simplify" their life and instead of using

$$B_i = \max\{C_j \mid \forall \tau_j, P_i < P_j\}$$

used

$$B_i = \max\{C_j \mid \forall \tau_j\}$$



Real-Time Systems
April 2007, Volume 35, Issue 3, pp 239–272 | Cite as

Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised

Authors | Authors and affiliations

Robert I. Davis, Alan Burns, Reinder J. Bril, Johan J. Lukkien

Open Access | Article
First Online: 30 January 2007

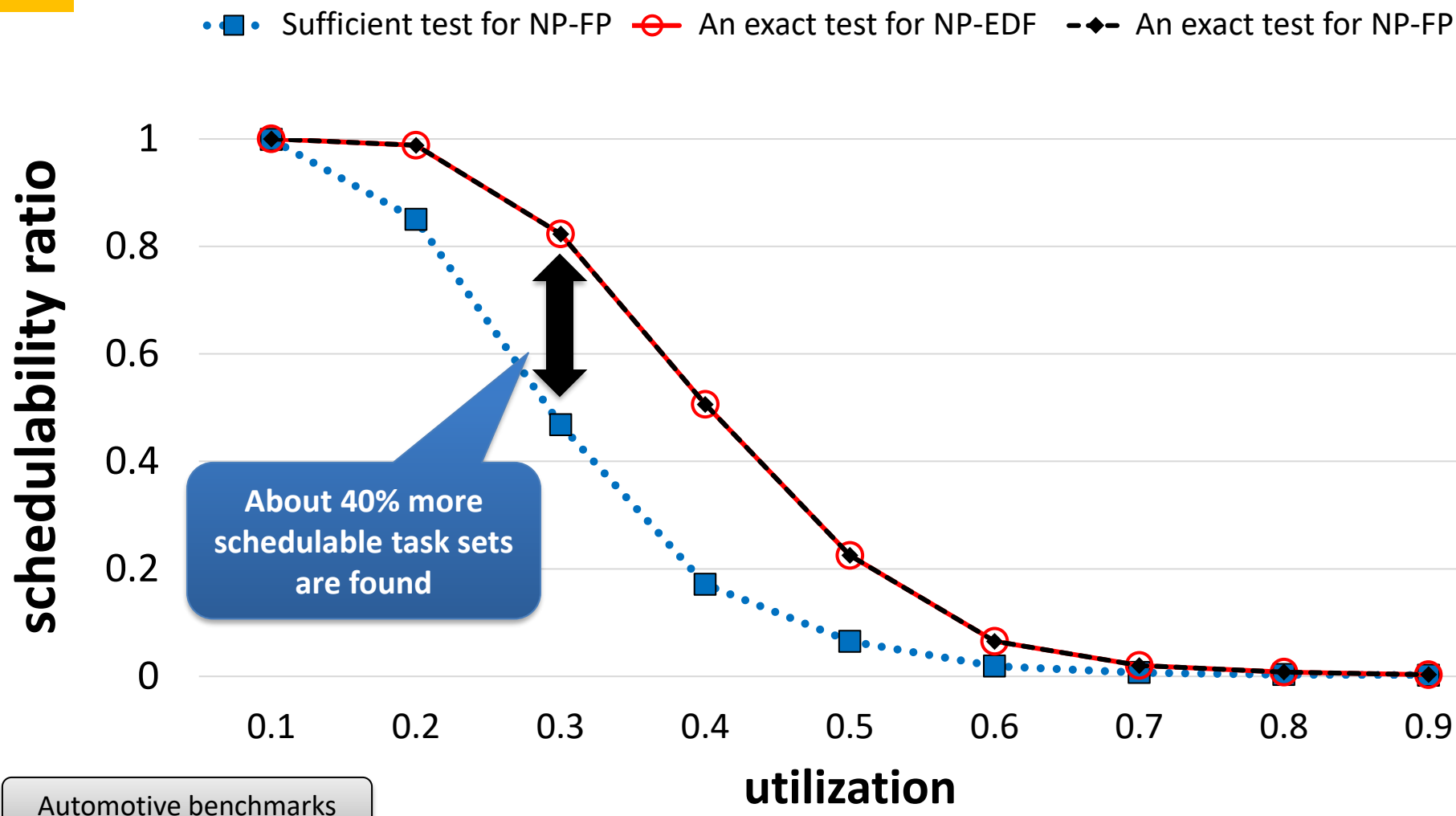
32 Shares | 9.7k Downloads | 325 Citations

Abstract

Controller Area Network (CAN) is used extensively in automotive applications, with in excess of 400 million CAN enabled microcontrollers manufactured each year. In 1994 schedulability analysis was developed for CAN, showing how worst-case response times of CAN messages could be calculated and hence guarantees provided that message response times would not exceed their deadlines. This seminal research has been cited in over 200 subsequent papers and transferred to industry in the form of commercial CAN schedulability analysis tools. These tools

Sufficient v.s. exact tests for NP-FP (and NP-EDF)

for periodic tasks



Automotive benchmarks

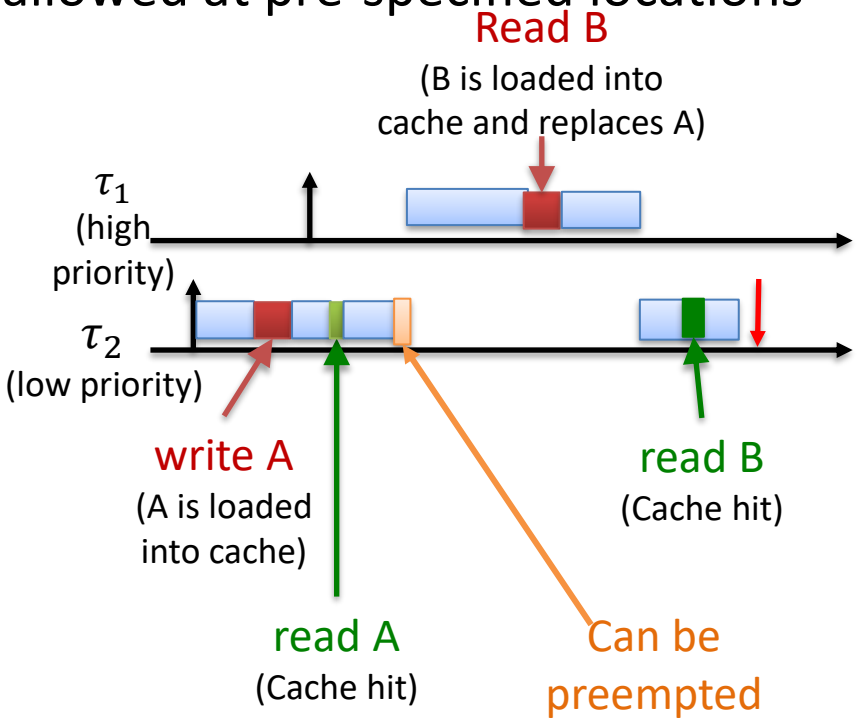
If you are interested:

An exact analysis for non-preemptive scheduling: Schedule Abstraction Graph

Mitra Nasri and Björn B. Brandenburg, "An Exact and Sustainable Analysis of Non-Preemptive Scheduling", in the Proceedings of the Real-Time Systems Symposium (RTSS), 2017, pp. 1-12.

Limited-Preemptive Scheduling

- Preemption Thresholds: Only tasks above a certain priority can preempt
- Deferred Preemptions: Each task has a interval of time where it executes non-preemptively
- Task Splitting: Preemption is only allowed at pre-specified locations inside the task



Limited-Preemptive Scheduling

- Preemption Thresholds: Only tasks above a certain priority can preempt
- Deferred Preemptions: Each task has a interval of time where it executes non-preemptively
- Task Splitting: Preemption is only allowed at pre-specified locations inside the task

Benefits

Reduction in preemption overheads

Reduction in blocking by lower priority tasks

Summary

- Disadvantages of preemptive scheduling
- Advantages and Disadvantages of non-preemptive scheduling
- Why is non-preemptive scheduling difficult?
- Two online non-work conserving scheduling policies
- Why analyzing non-preemptive schedules is difficult?