Semi-Partitioned Scheduling of Dynamic Real-Time Workload:

A Practical Approach Based On Analysis-driven Load Balancing

Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo

Scuola Superiore Sant'Anna – ReTiS Laboratory

Pisa, Italy

This talk in a nutshell

Linear-time methods for **task splitting**

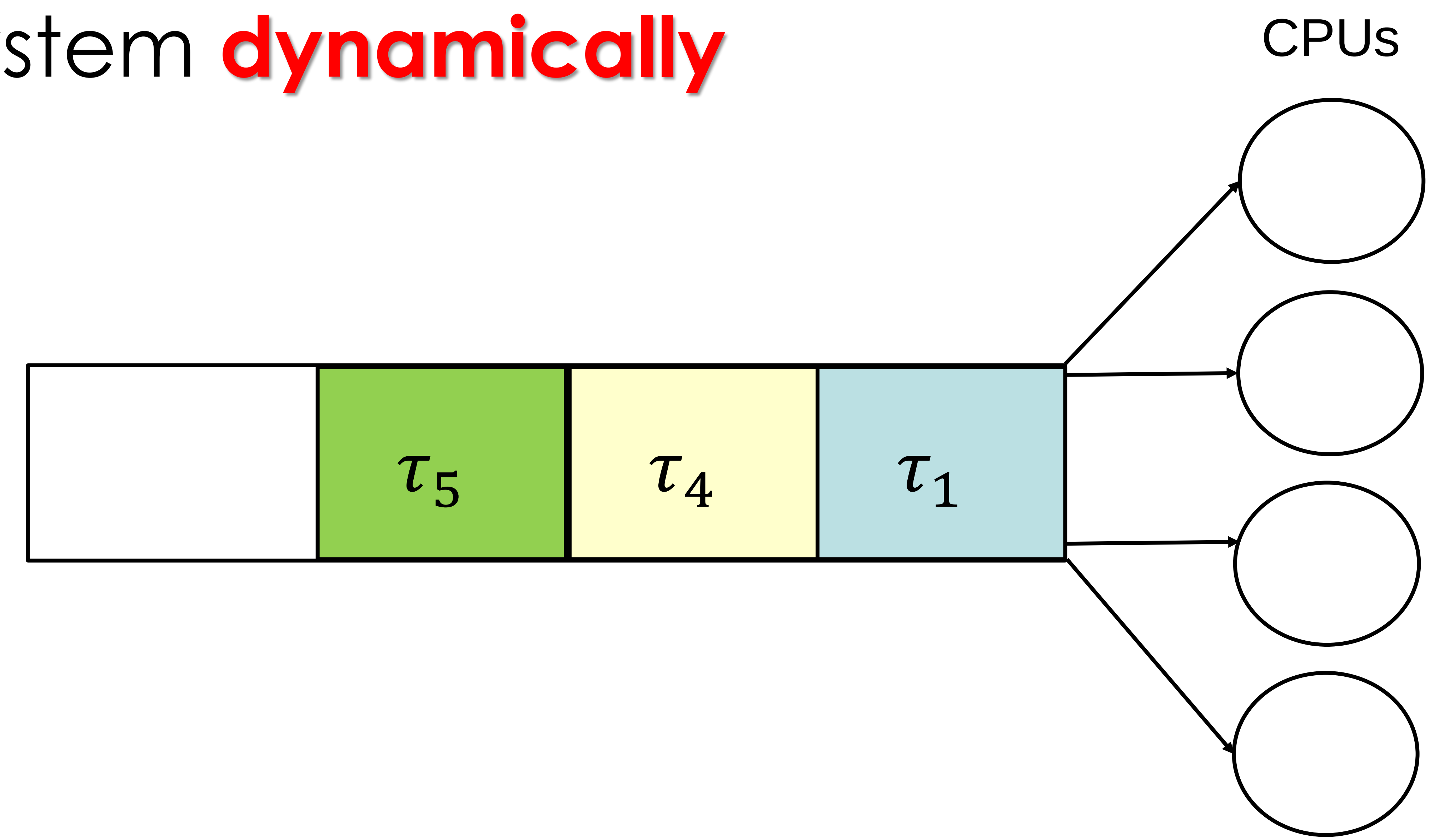
Approximation scheme for C=D with very limited utilization loss **(<3%)**

Load balancing algorithms for **semi-partitioned** scheduling

How to handle dynamic workload under semi-partitioned scheduling with *limited task re-allocations* and high schedulability performance **(>87%)**

Dynamic real-time workload

- Real-time tasks can **join** and **leave** the system **dynamically**



No **a-priori knowledge** of the workload

Is dynamic workload relevant?

- Many real-time applications do not have **a-priori** knowledge of the workload
  - Cloud computing, multimedia, real-time databases,...

- Example: multimedia applications with Linux that require guaranteed timing performance
  - Workload typically changes at runtime while the system is operating
  - SCHED_DEADLINE scheduling class can be used to achieve EDF scheduling with reservations

Is dynamic workload relevant?

- Many real-time **operating systems** provide syscalls to *spawn* tasks at run-time

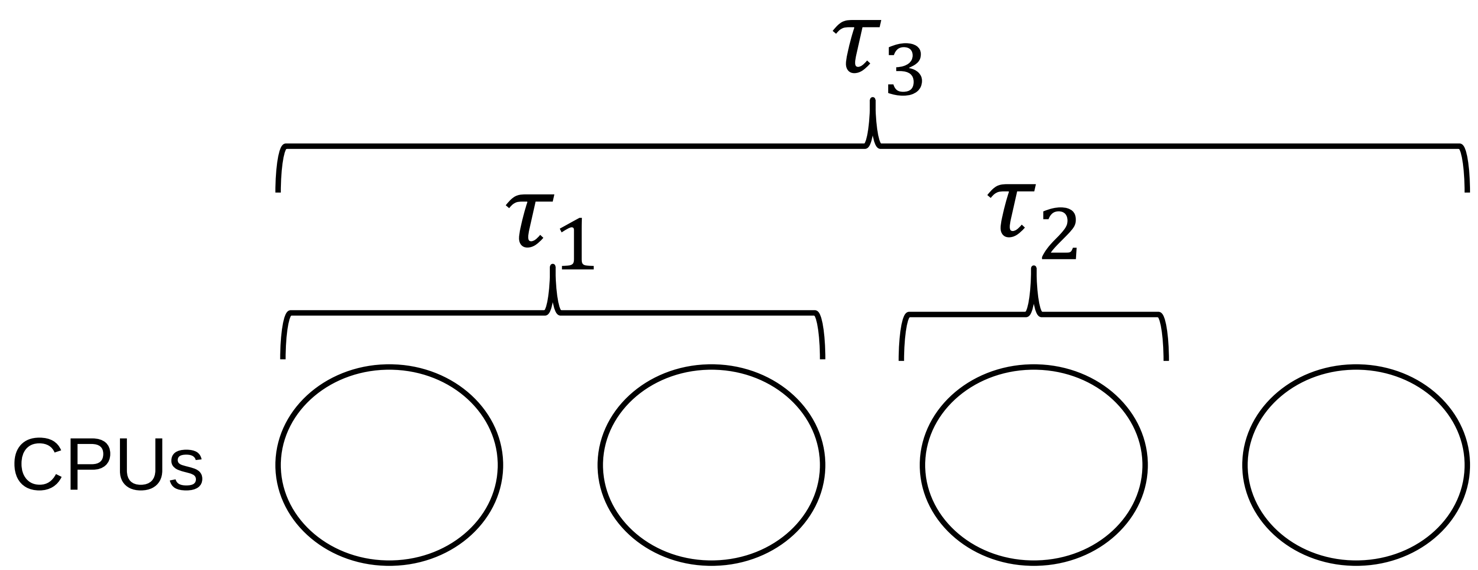
Linux
(SCHED_DEADLINE)

VxWorks

QNX™

Multiprocessor Scheduling

- Most RTOSes for multiprocessors implement APA (Arbitrary Processor Affinities) schedulers

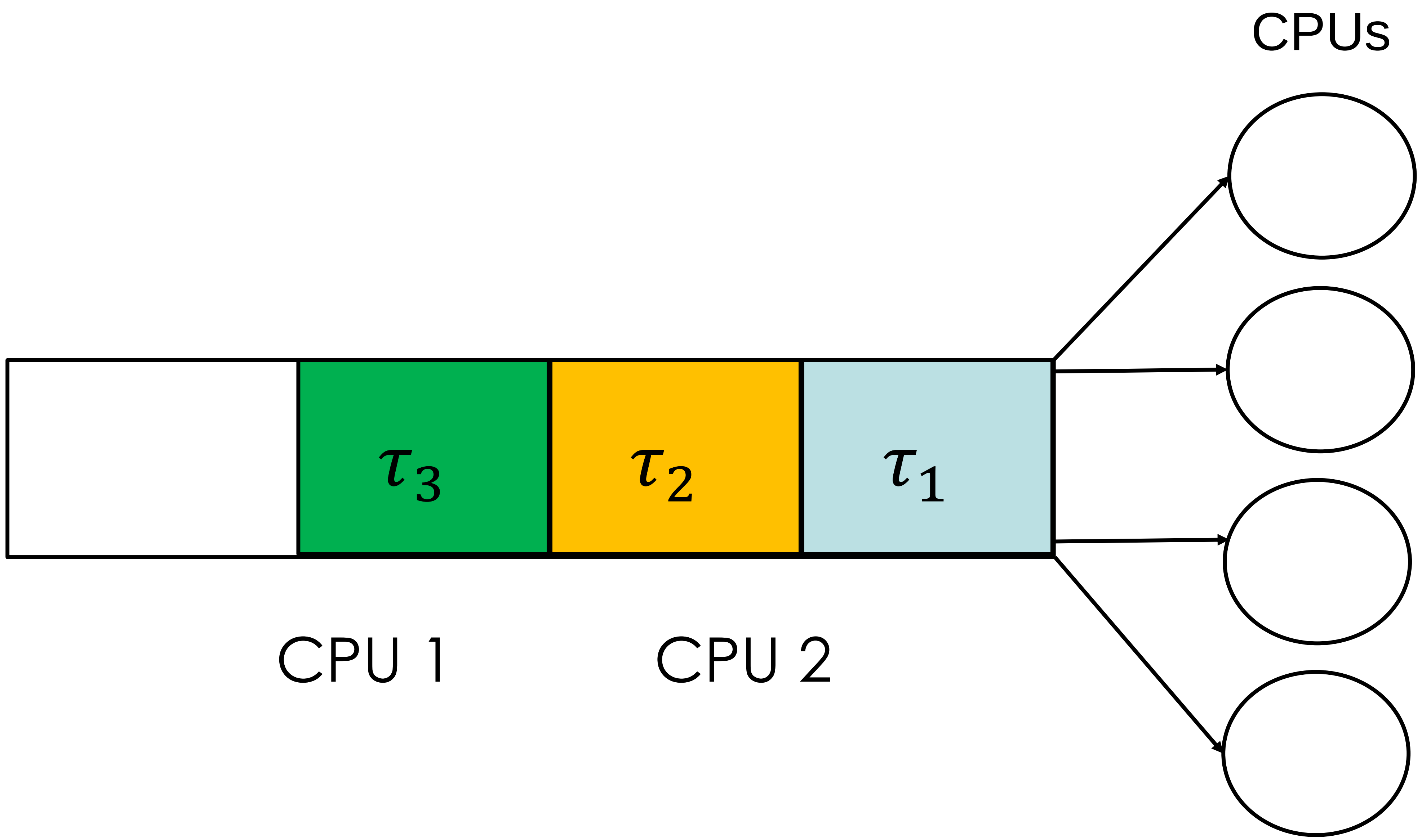


Global Scheduling

Partitioned Scheduling

Global Scheduling

Provides automatic load-balancing (*transparent*) by construction



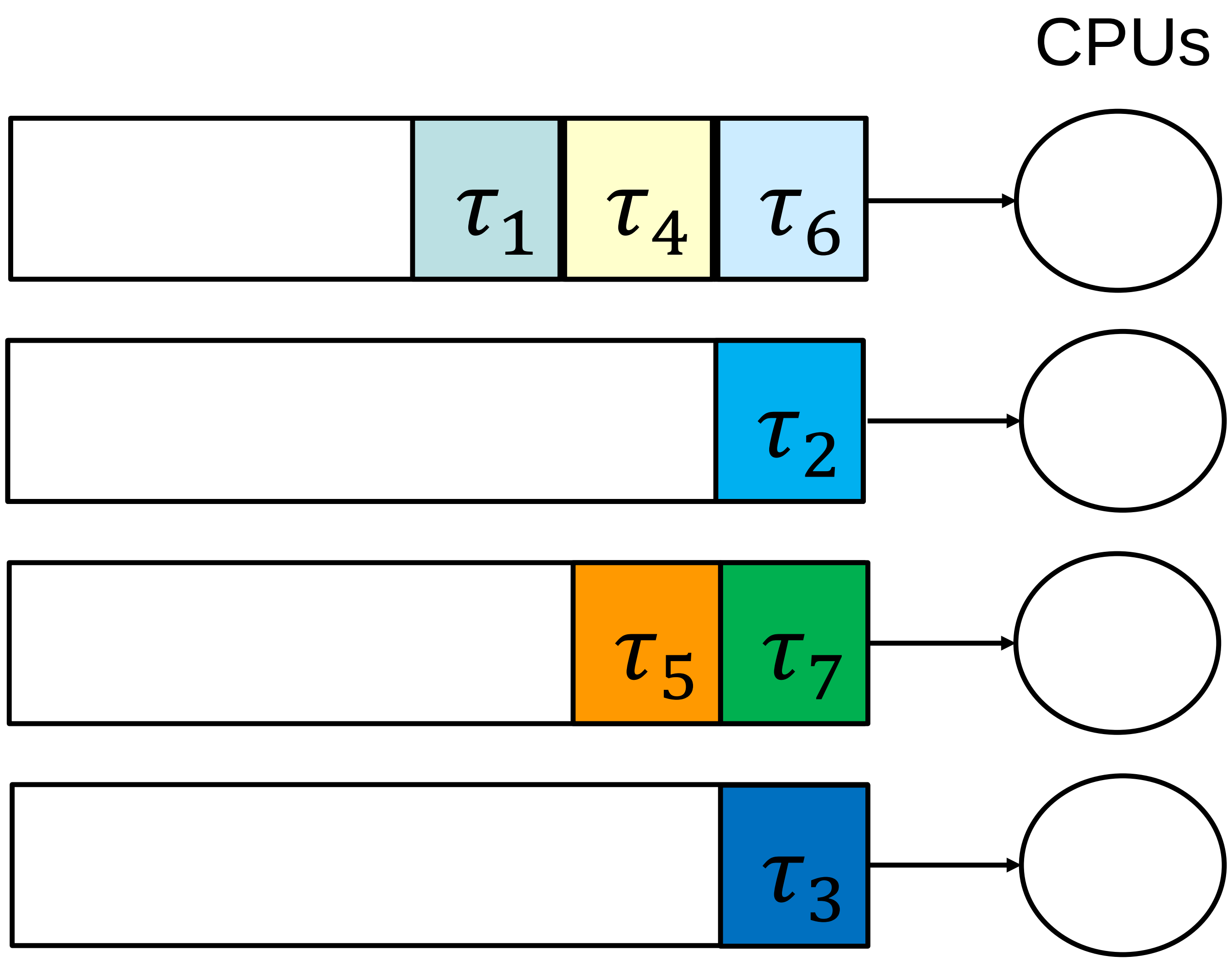
Global Scheduling

- ✔ Automatic load balancing
- ✘ High run-time overhead
- ✘ Execution difficult to predict
- ✘ Difficult derivation of worst-case bounds

…

Partitioned Scheduling

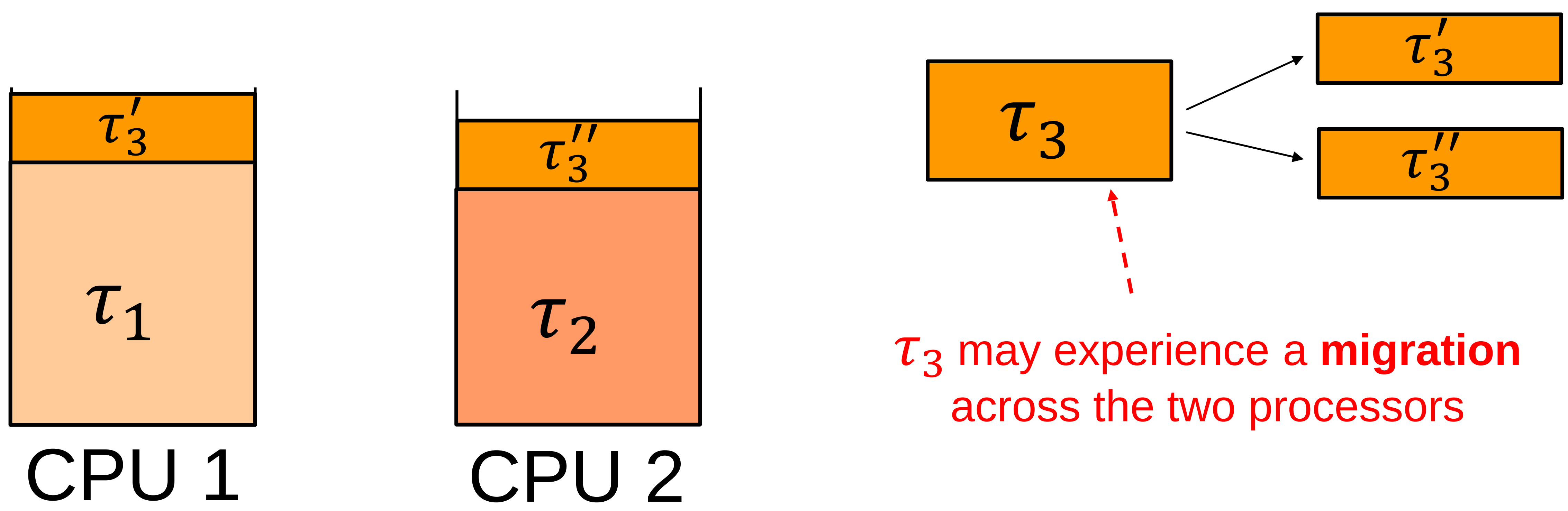
Typically exploits **a-priori** knowledge
of the workload and an **off-line** partitioning phase



Semi-Partitioned Scheduling

Anderson et al. (2005)

- Builds upon partitioned scheduling
- Tasks that do not fit in a processor are **split** into **sub-tasks**



C=D Splitting

Burns et al. (2010)

- Allows to split tasks into multiple chunks, with the first n-1 chunks at zero-laxity (C = D)
- Based on EDF

Example: two chunks

$$\tau_3 = (C_i, D_i, T_i) = (30, 100, 100)$$

$$\tau_3' = (20, 20, 100)$$

Zero-laxity chunk

$$\mathrm{C_i} = \mathrm{D_i}$$

$$\tau_3'' = (10, 80, 100)$$

Last chunk

$$\mathrm{D_i''} = \mathrm{T_i} - \mathrm{D_i'}$$

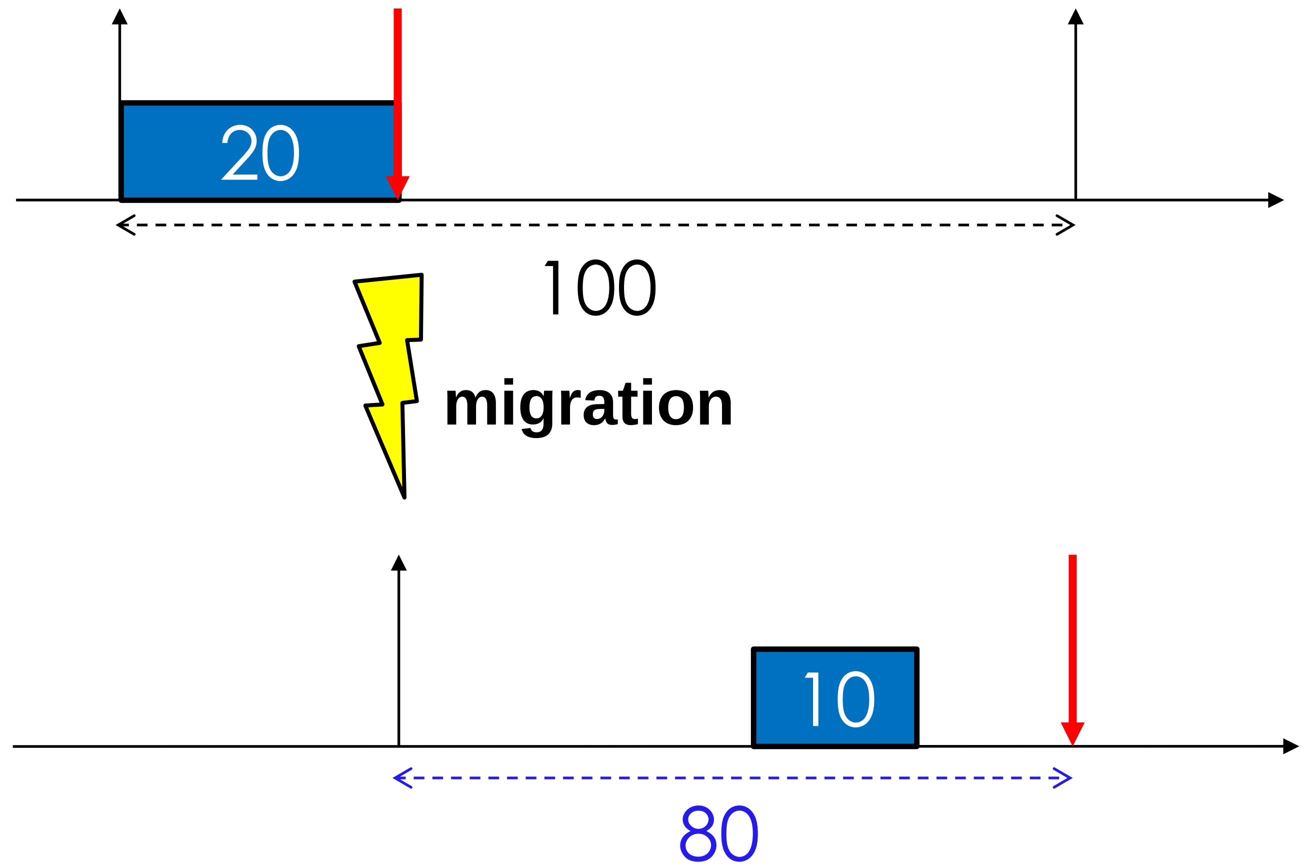
C=D Splitting

Burns et al. (2010)

- Allows to split tasks into multiple chunks, with the first n-1 chunks at zero-laxity (C = D)
- Based on EDF

$$\tau_3' = (20, 20, 100)$$

$$\tau_3'' = (10, 80, 100)$$



A very important result

Brandenburg and Gül (2016)

"Global Scheduling Not Required"

Empirically, near-optimal schedulability (99%+) achieved with simple, well-known and low-overhead techniques

- Based on C=D Semi-Partitioned Scheduling
- Performance achieved by applying multiple clever heuristics (off-line)

Conceived for **static** workload

Semi-Partitioned Scheduling

- ✓ More predictable execution
- ✓ Reuse of results for uniprocessors
- ✓ Excellent worst-case performance
- ✓ Low overhead
- ✗ A-priori knowledge of the workload
- ✗ Off-line partitioning and splitting phase

Global vs Semi-partitioned

Global

- ✔ Automatic load balancing
- ✘ High run-time overhead
- ✘ Execution difficult to predict
- ✘ Difficulty in deriving worst-case bounds

Semi-Partitioned

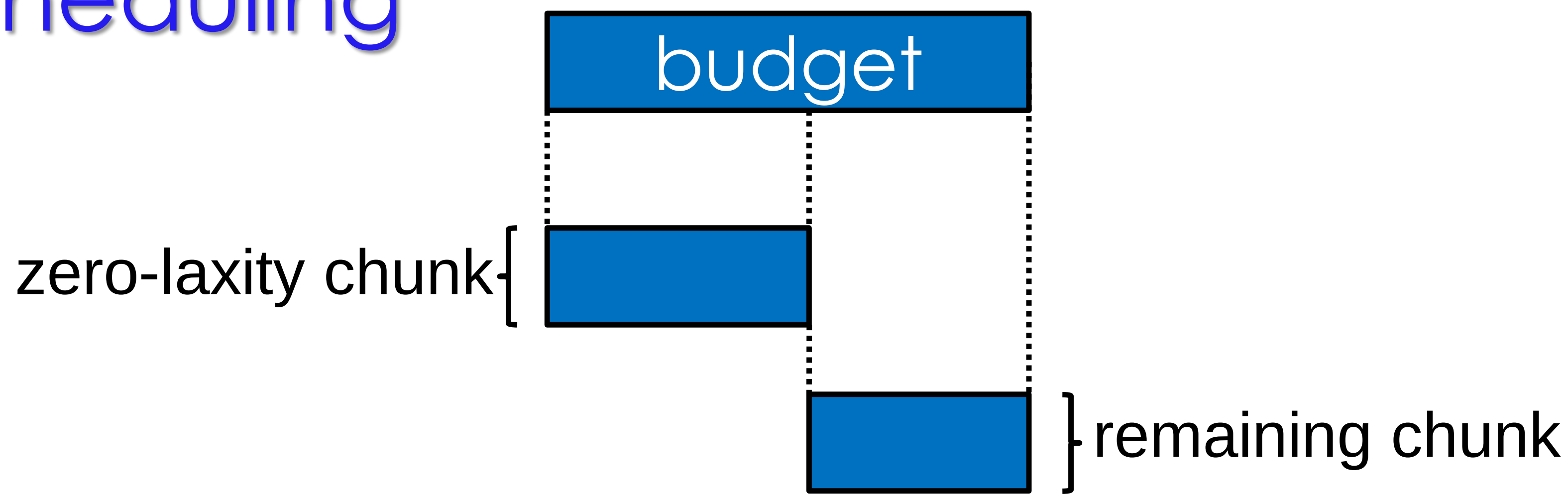
- ✔ More predictable execution
- ✔ Reuse of results of uniprocessors
- ✔ Excellent worst-case performance
- ✔ Low overhead
- ✘ Off-line partitioning and splitting phase
- ✘ A-priori knowledge of the workload

HOW TO MAINTAIN THE BENEFITS OF SEMI-PARTITIONED SCHEDULING WITHOUT REQUIRING ANY OFF-LINE PHASE?

How to partition and split tasks online?

This work

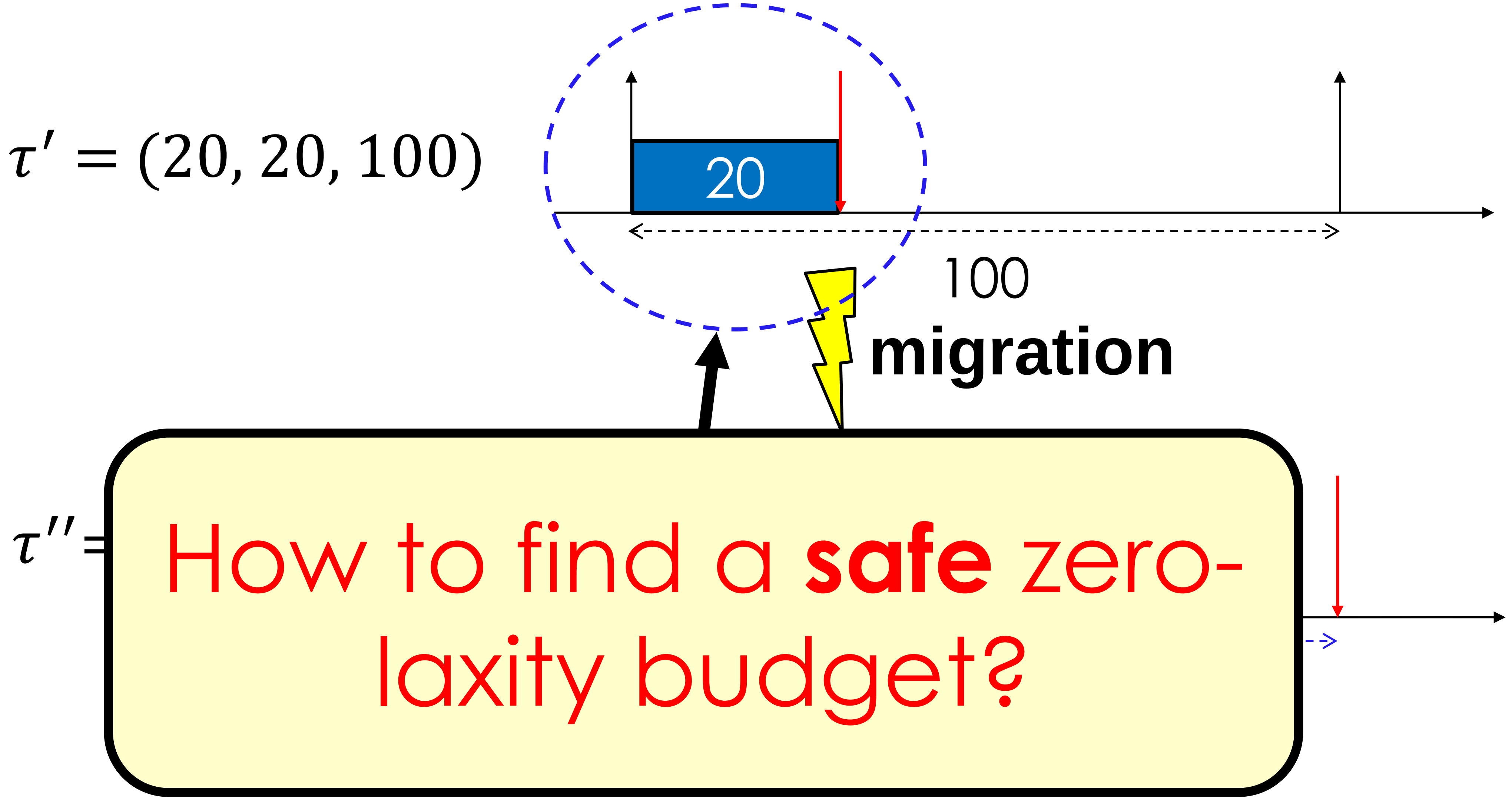
- This work considers dynamic workload consisting of reservations (budget, period)
  - The consideration of this model is compliant with the one available in Linux (SCHED_DEADLINE), hence present in billions of devices around the world
- The workload is executed under C=D Semi-Partitioned Scheduling
  - Budget splitting



C=D Budget Splitting

τ= (budget = 30, period = 100)

to be split

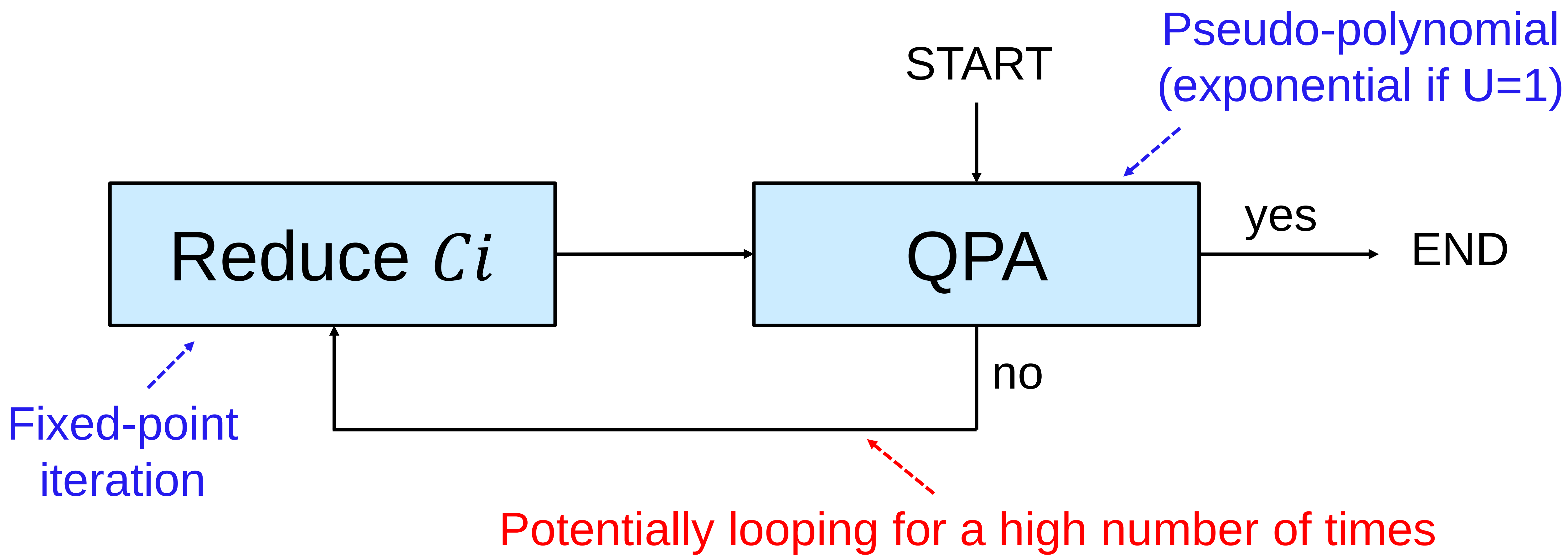


How to find the zero-laxity budget?

Burns et al. (2010)

- Iterative process based on **QPA** (*Quick Processor-demand Analysis*) with high complexity (no bound provided by the authors)

- Also used by Brandenburg and Gül (2016)



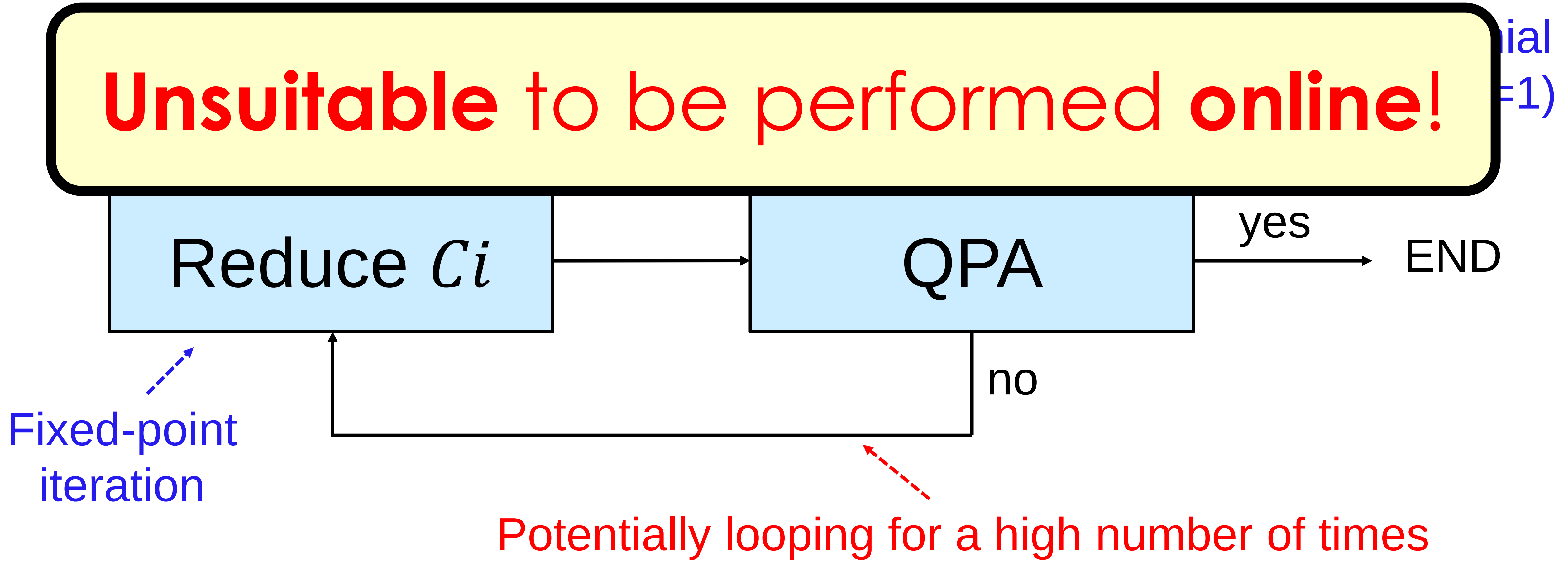
How to find the zero-laxity budget?

Burns et al. (2010)

- Iterative process based on **QPA** (*Quick Processor-demand Analysis*) with high complexity (no bound provided by the authors)

- Also used by Brandenburg and Gül (2016)

Unsuitable to be performed **online**!



Our approach: approximated C=D

Main goal: Compute a safe bound for the zero-laxity budget in linear time

- In this work we proposed an approximate method based on solving a system of inequalities

$$\begin{cases} C' = D' \leq K_1 \\ \dots \\ C' = D' \leq K_N \end{cases} \Rightarrow C' = \min(K_1, \dots, K_N)$$

Constants depending on static task-set parameters

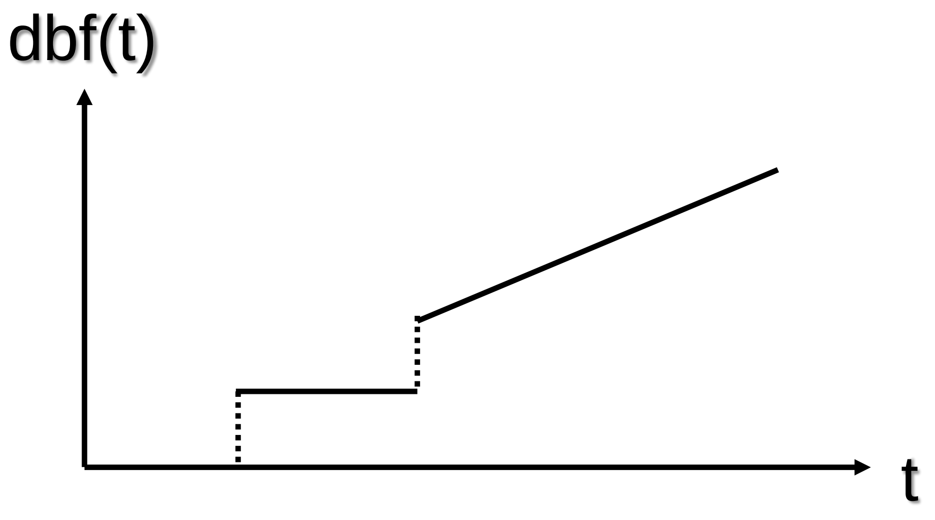
order of number of tasks

Our approach: approximated C=D

How have we achieved the closed-form formulation?

- Approach based on approximate demand-bound functions

  Some of them similar to those proposed by *Fisher et al.* (2006)



- + <u>theorems</u> to obtain a closed-form formulation

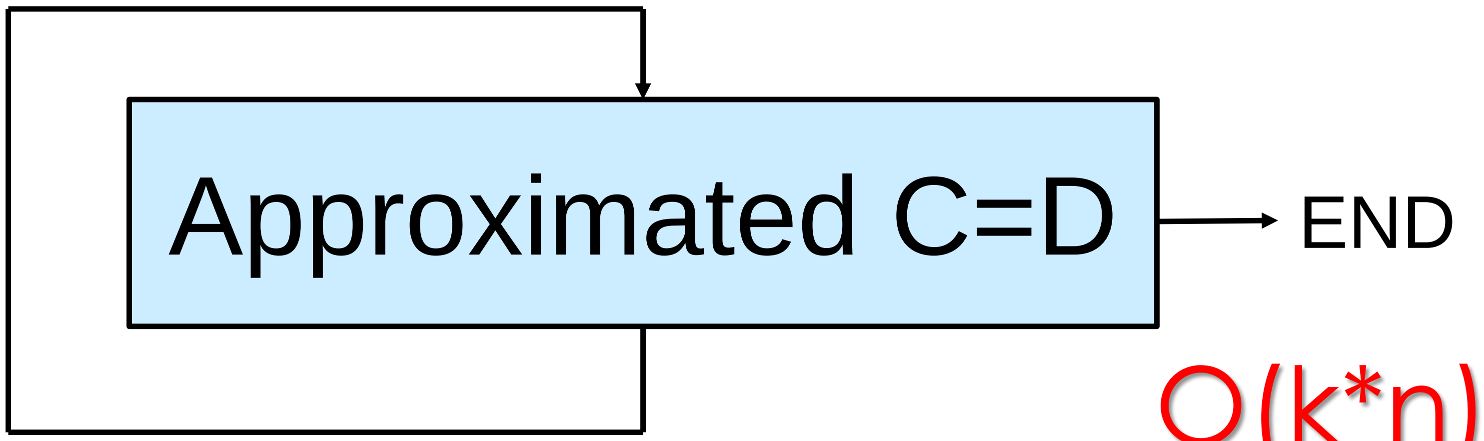
  The derivation of the closed-form solution has been also mechanized with the Wolfram Mathematica tool

Approximated C=D: Extensions

The approximation can be improved by:

- **Extension 1:** Iterative algorithm that refines the bound

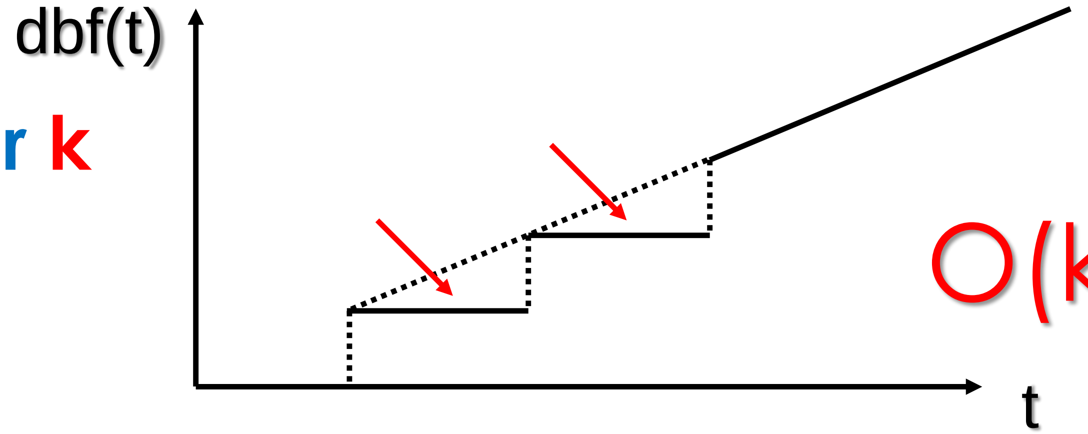
Repeats for a fixed number **k** of refinements



O(k*n)

- **Extension 2:** Refinement on the precisions of the approximate dbfs

Add a fixed number **k** of discontinuities

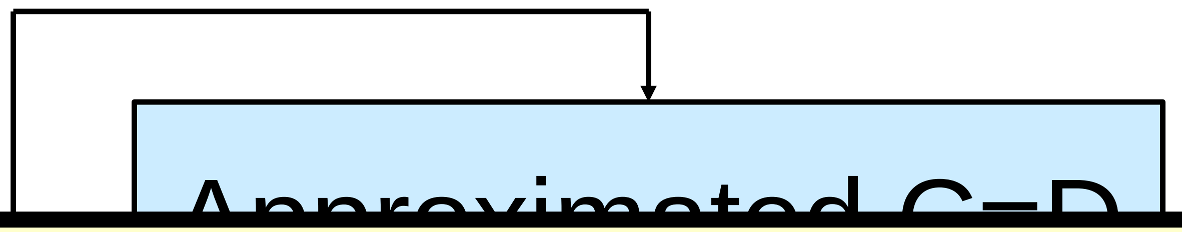


O(k*n)

Approximated C=D: Extensions

The approximation can be improved by:

- **Extension 1:** Iterative algorithm that refines the bound

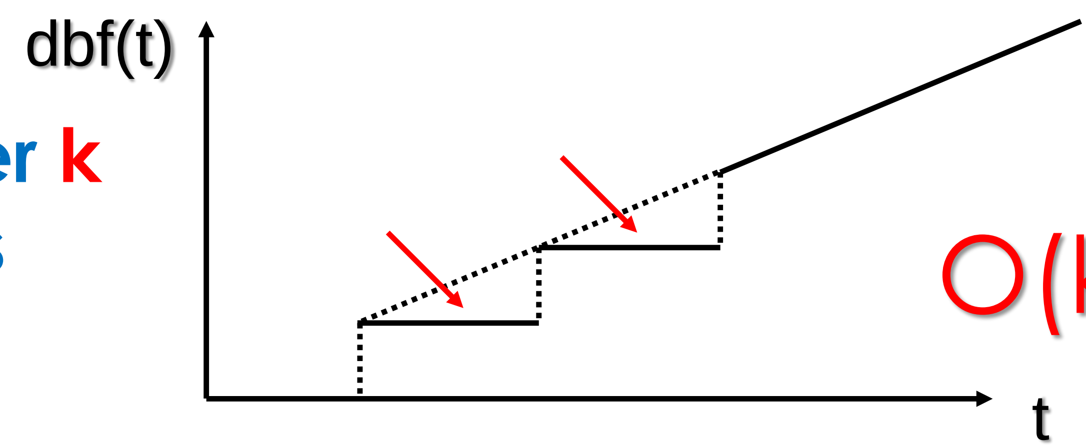


Repeats for a fixed

We found that significant improvements can be achieved with just two iterations

- **Extension 2:** Refinement on the precisions of the approximate dbfs

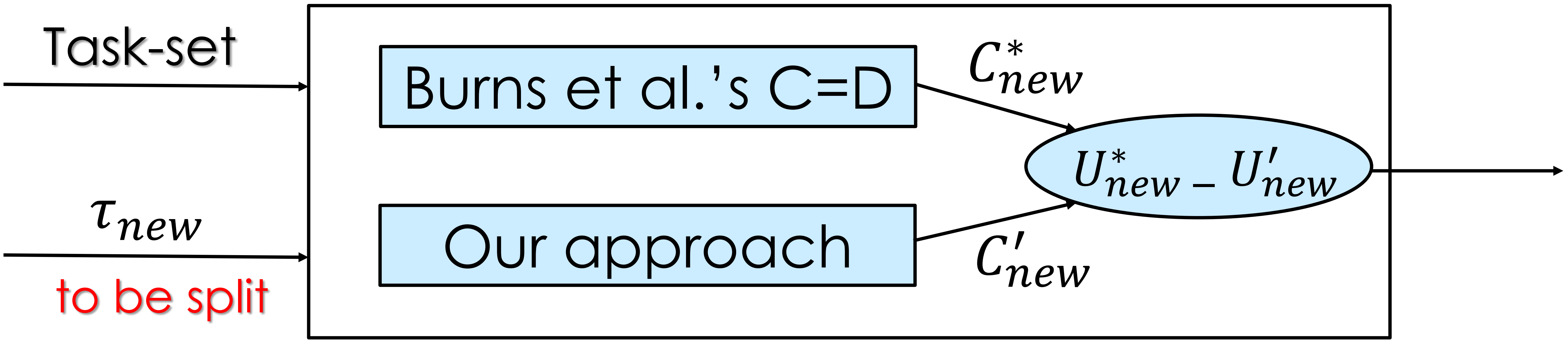
Add a fixed number k of discontinuities



O(k*n)

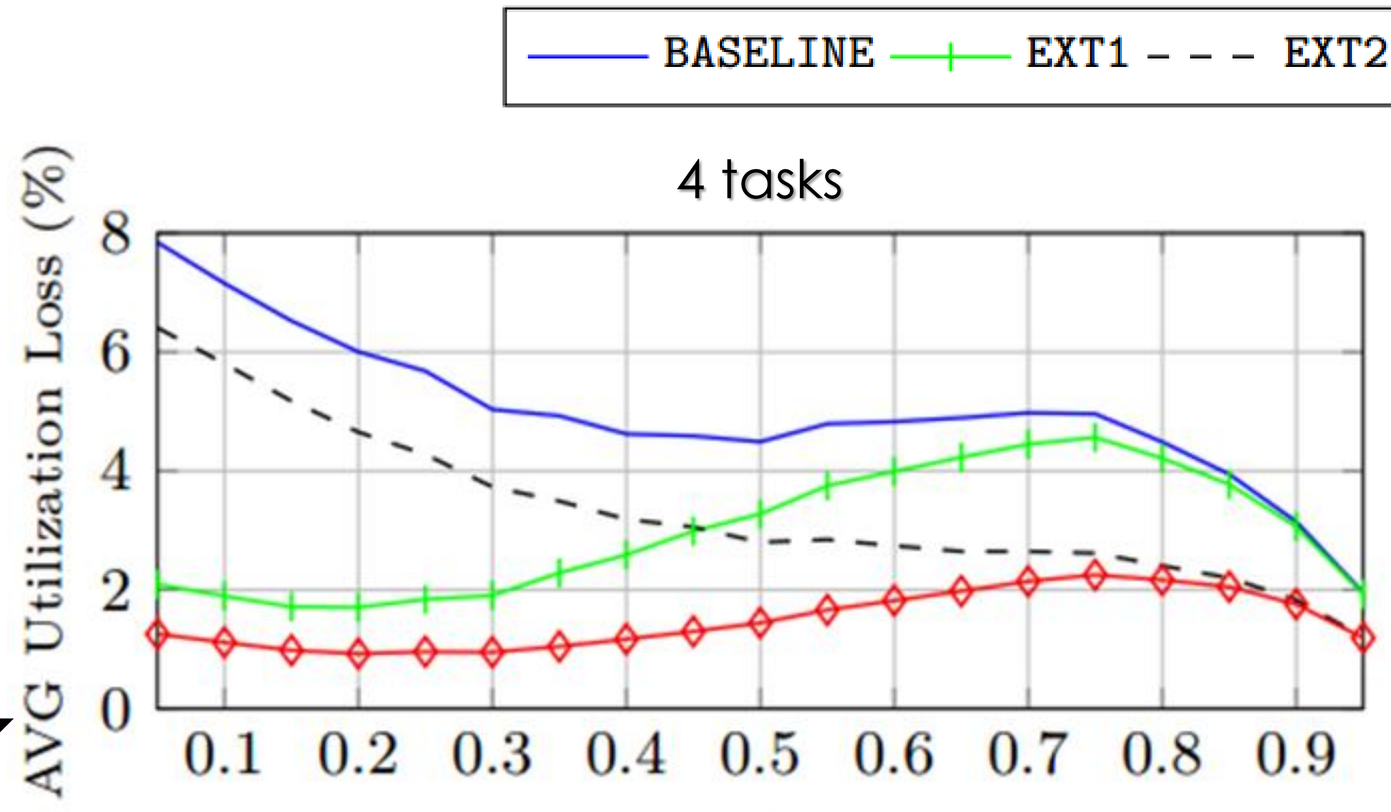
Experimental Study

- Measure the utilization loss introduced by our approach with respect to the (exact) Burns et al.'s algorithm



- Tested almost 2 Million of task sets over wide range of parameters

Representative Results



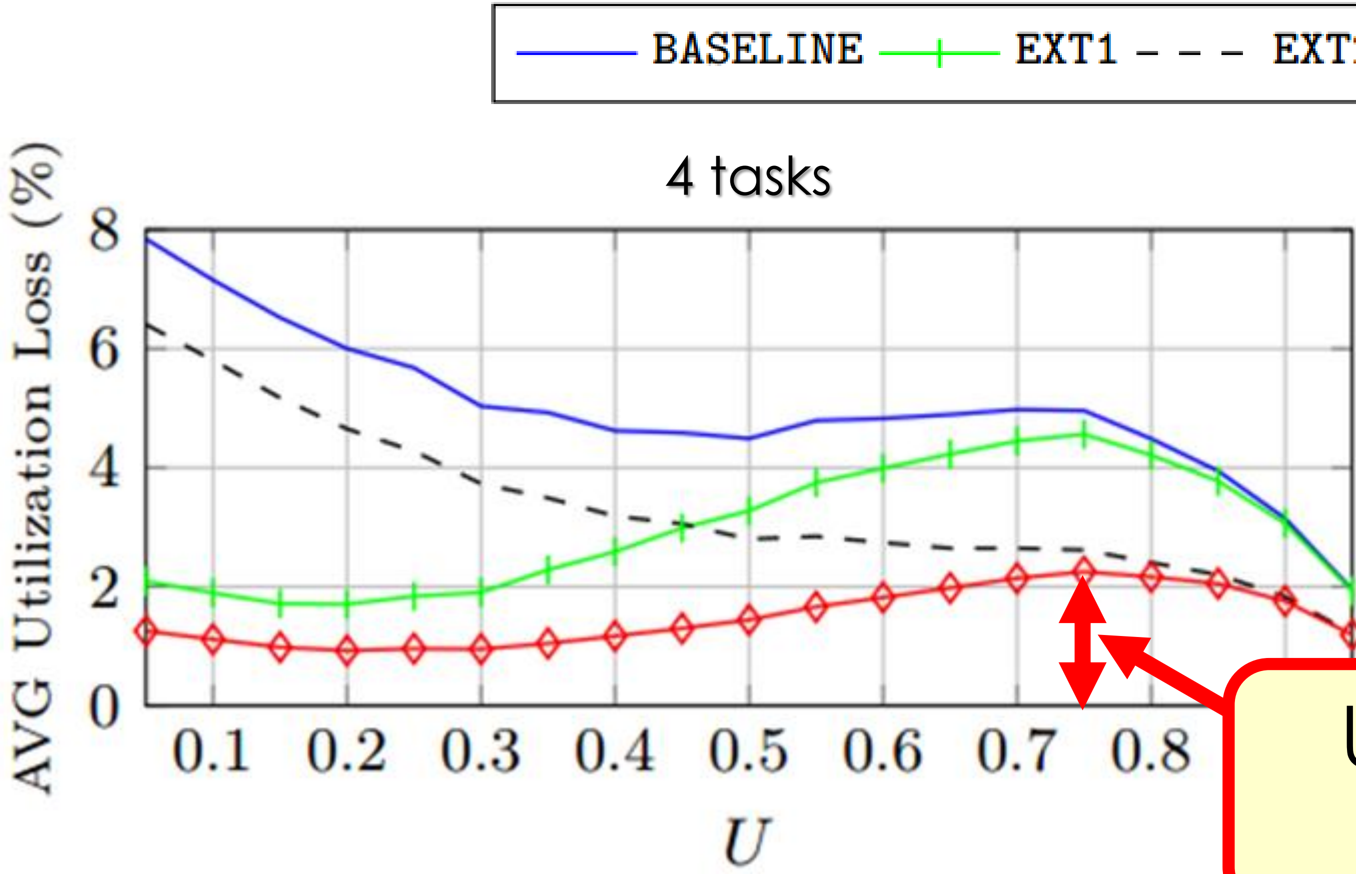
The lower the better

Increasing CPU load

Extension 1 is effective for low utilization values

Extension 2 is effective for high utilization values

Representative Results

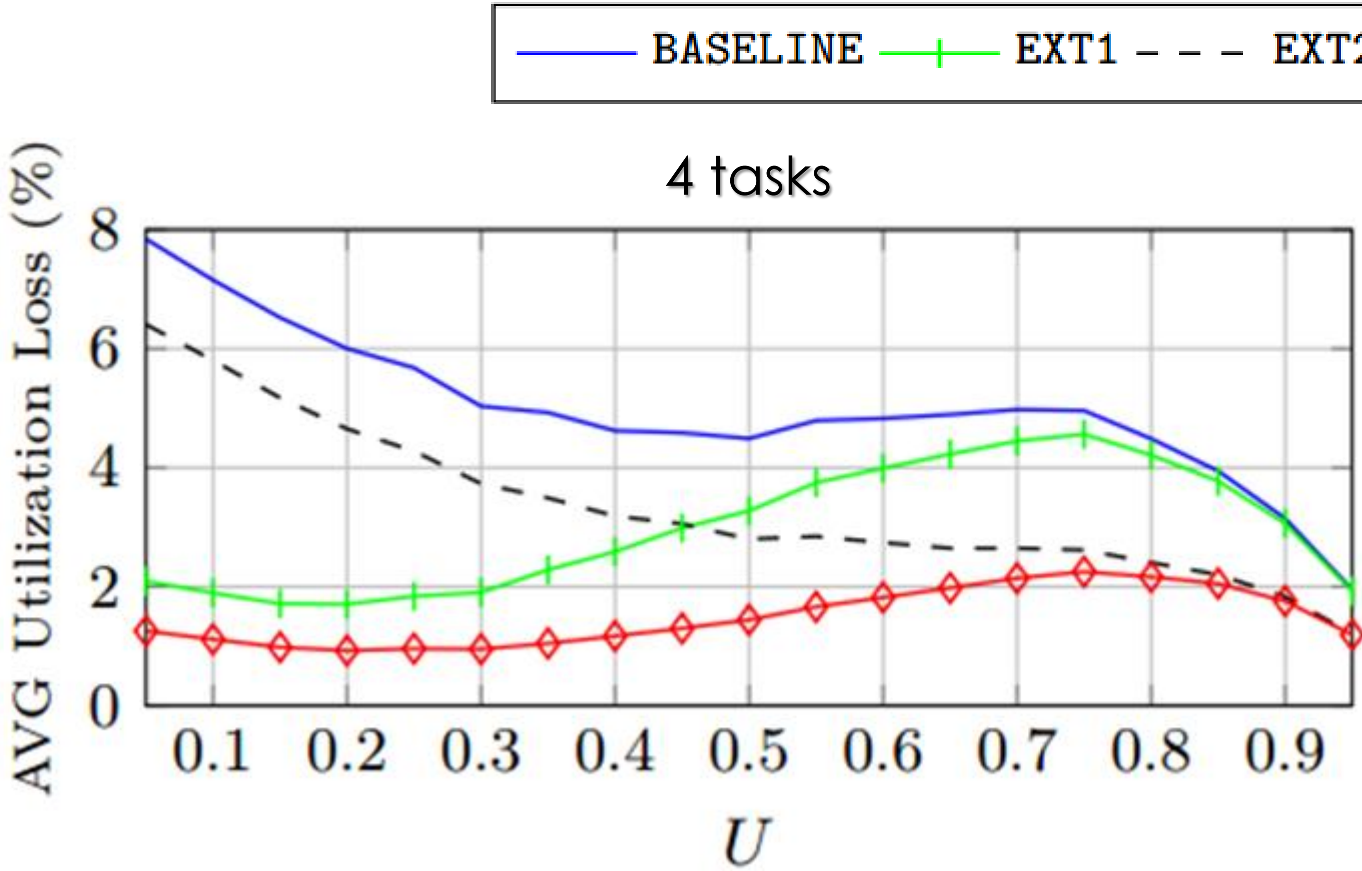


Extension 1 is effective for low utilization values

Extension 2 is effective for high utilization values

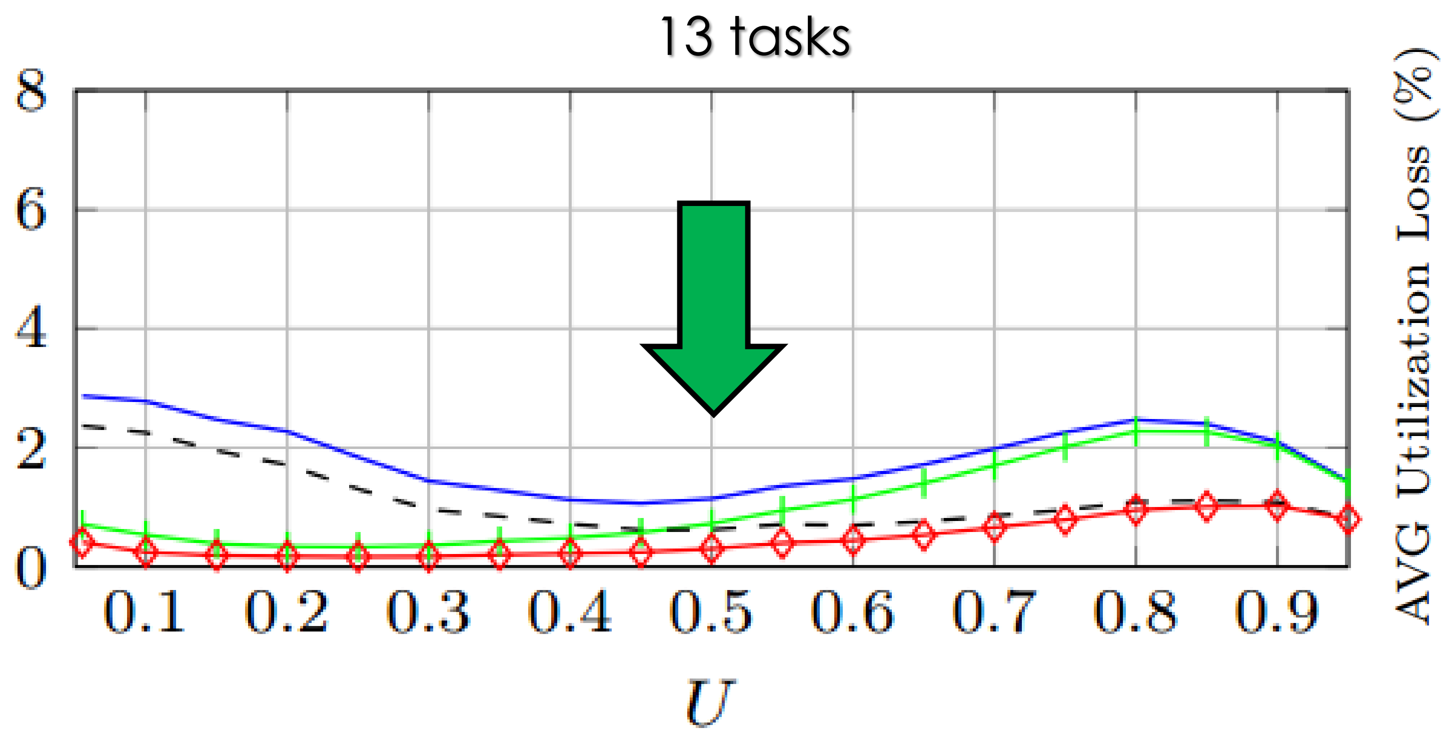
Utilization loss **~2%** w.r.t. the exact algorithm

Representative Results



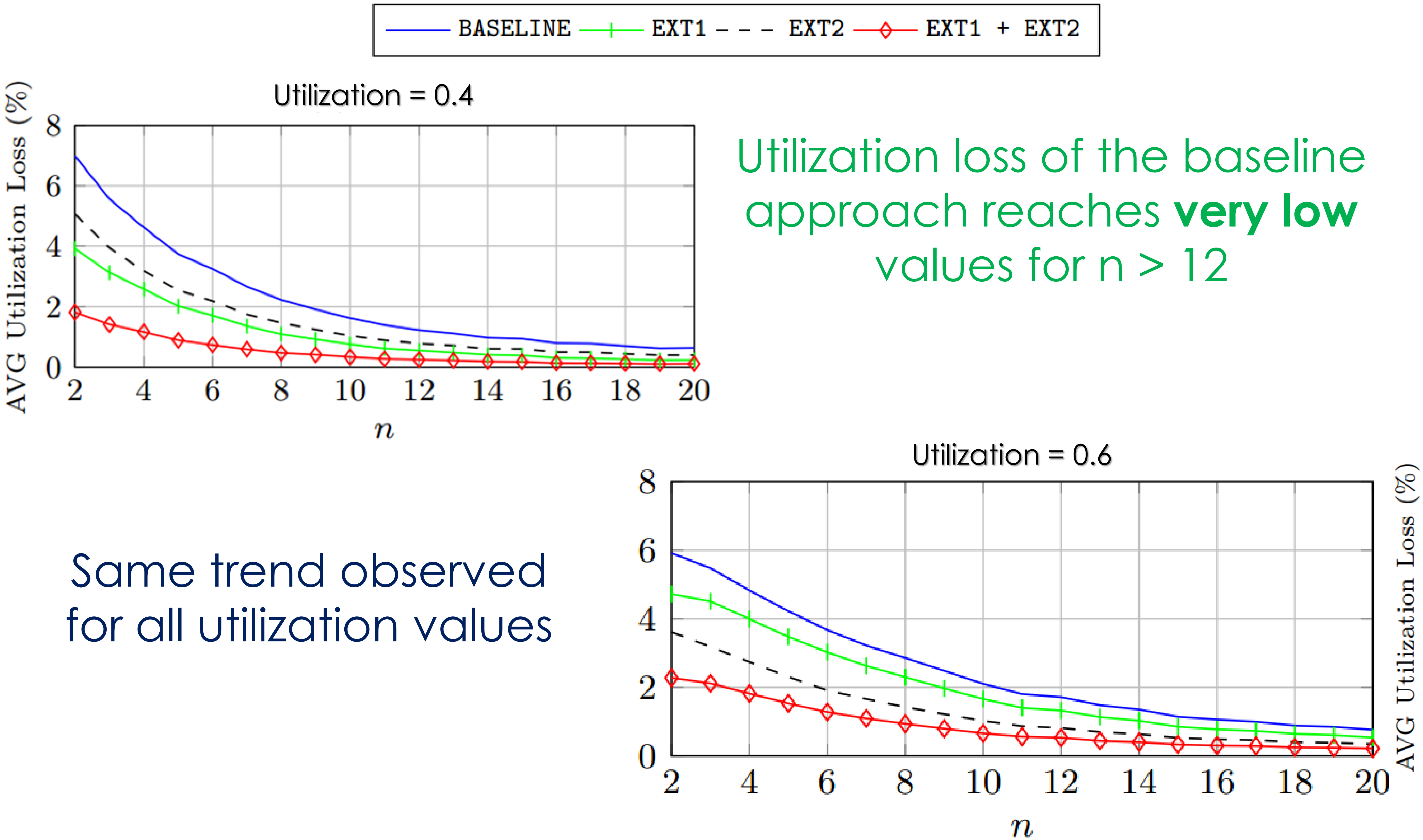
Extension 1 is effective for low utilization values

Extension 2 is effective for high utilization values



The average utilization loss decreases as the number of tasks increases

Representative Results



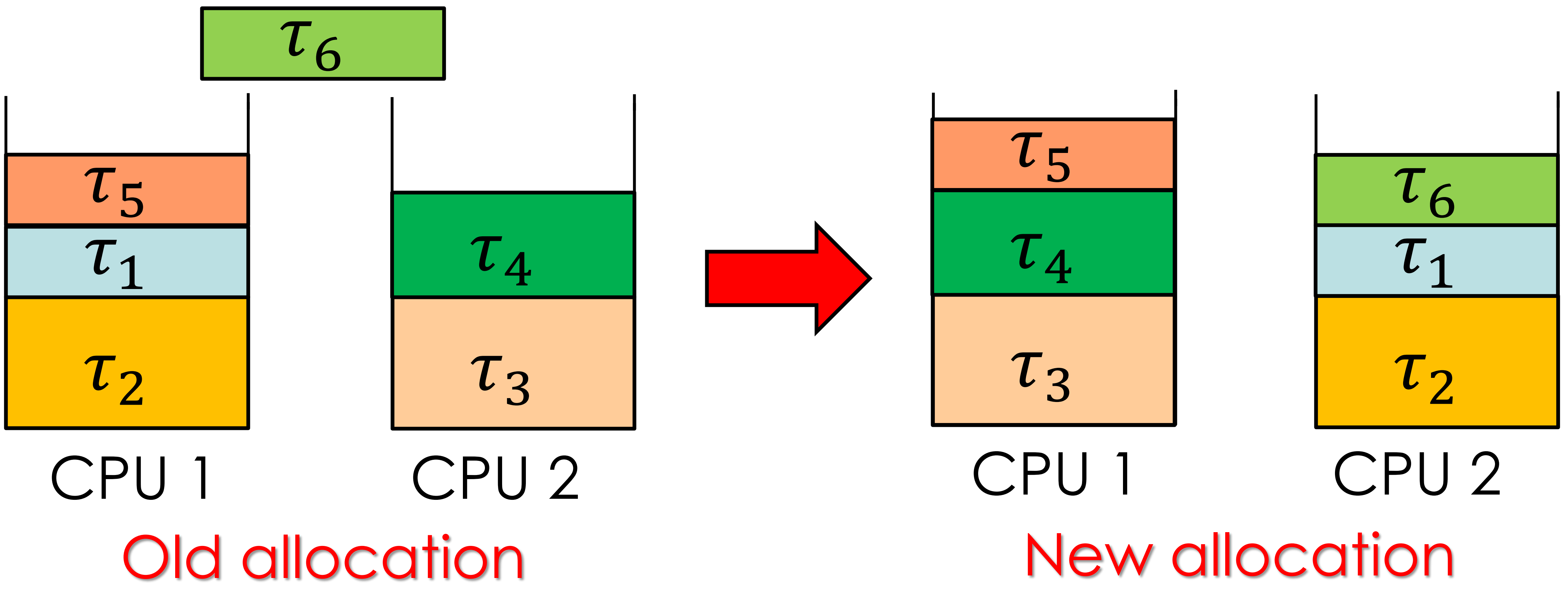
Utilization loss of the baseline approach reaches **very low** values for n > 12

Same trend observed for all utilization values

HOW TO APPLY ON-LINE SEMI-PARTITIONING TO PERFORM LOAD BALACING?

Why do not use classical approaches?

- Existing task-placement algorithms for semi-partitioning would require **reallocating** many tasks (they were conceived for static workload)



Impracticable to be performed on-line:
the previous allocation cannot be ignored!

The problem

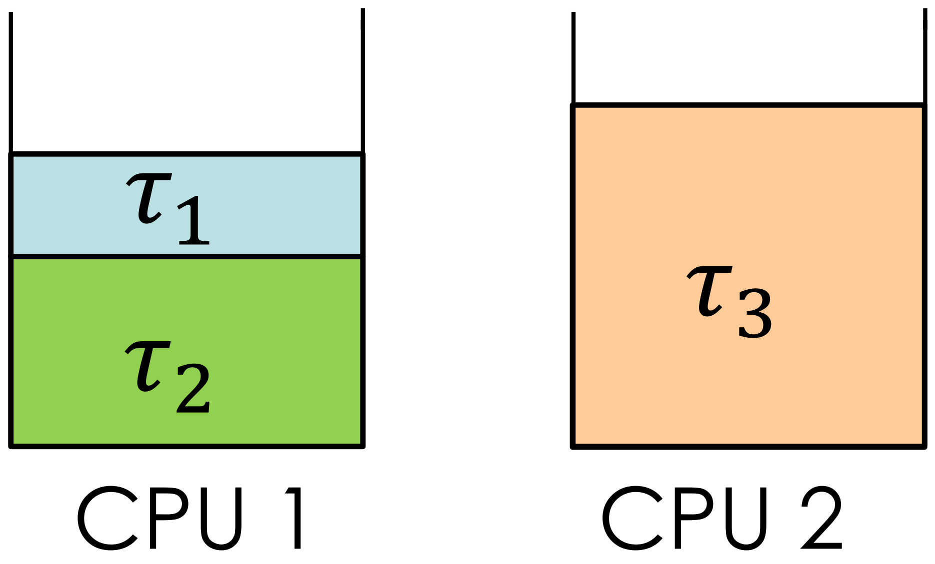
How to achieve high schedulability performance with

- a very limited number of re-allocations; and
- keeping the mechanism as simple as possible?

Focus on **practical applicability**

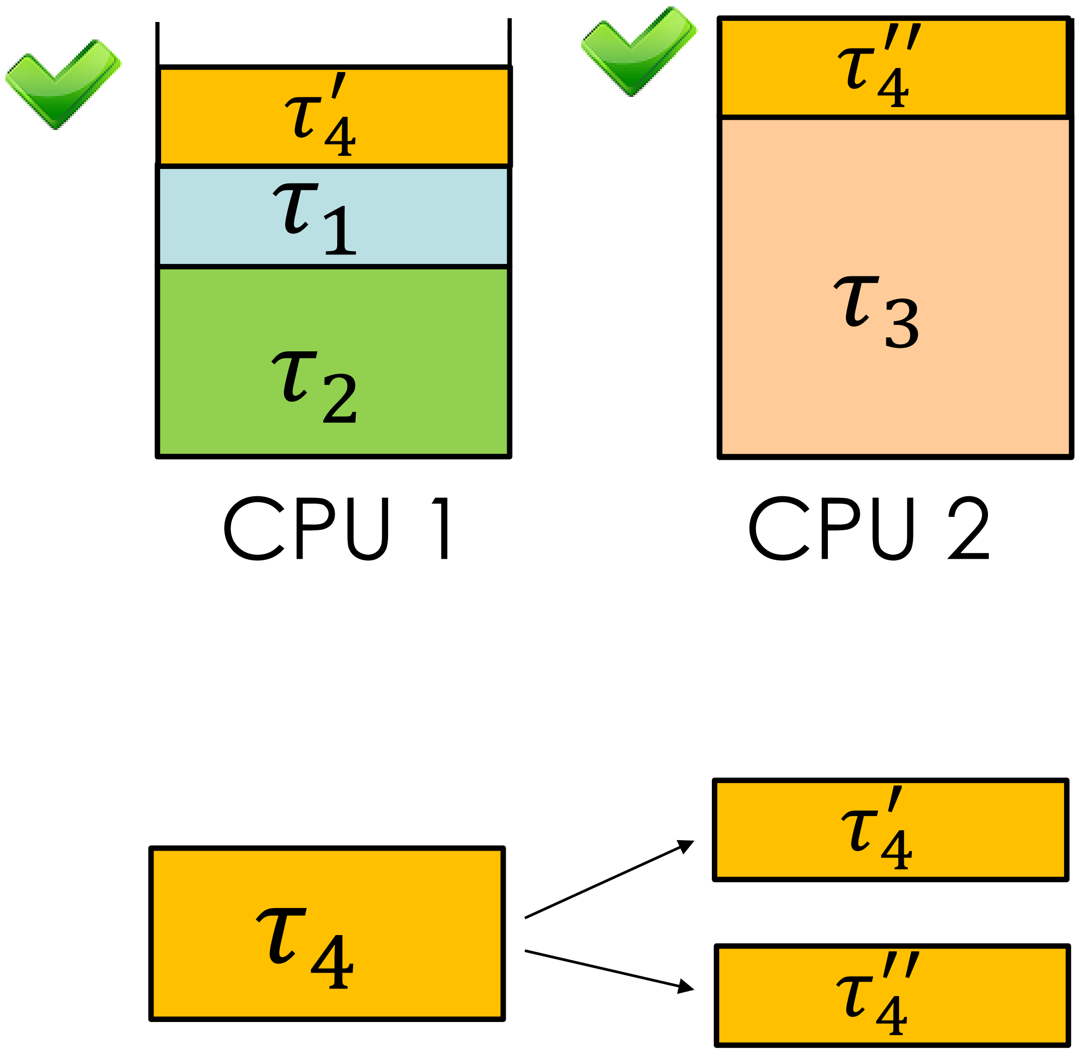
Proposed approach

First try a simple bin packing heuristics (e.g., first-fit)



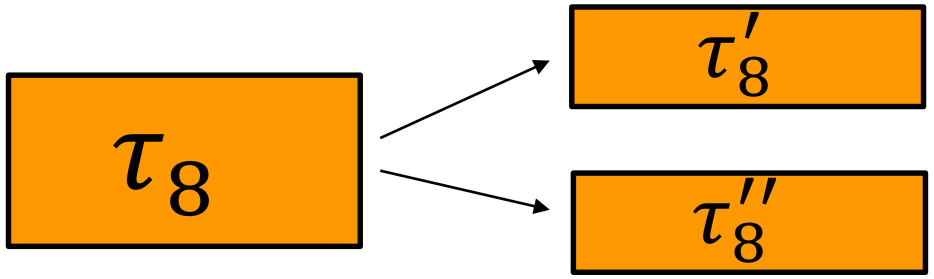
Proposed approach

If **not** schedulable, **try to split**



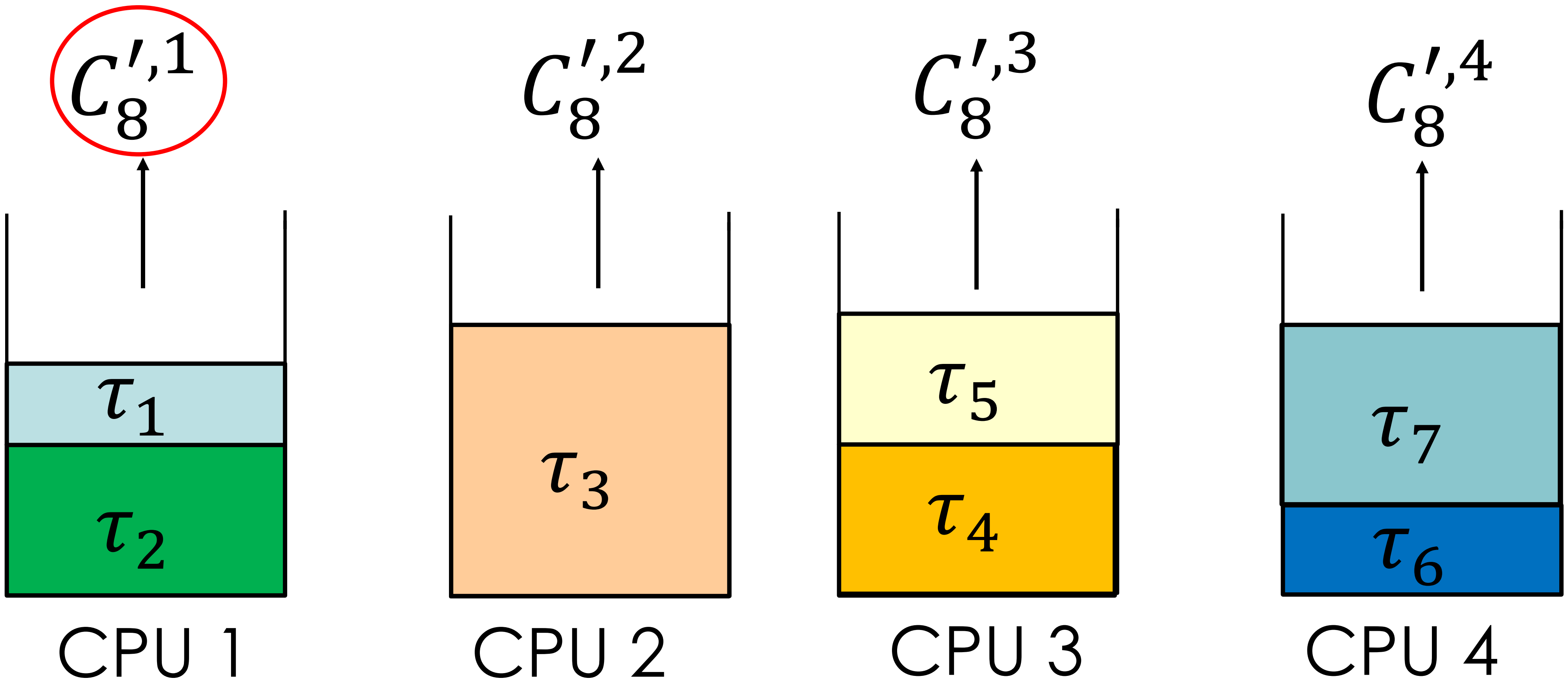
Proposed approach

- How to split?



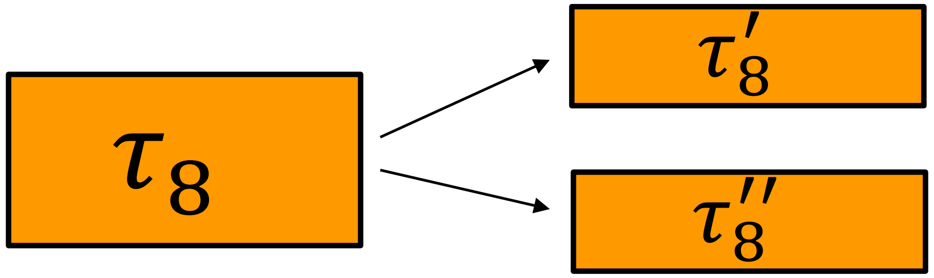
take the maximum zero-laxity budget across the processors

$$\max C_8'$$



Proposed approach

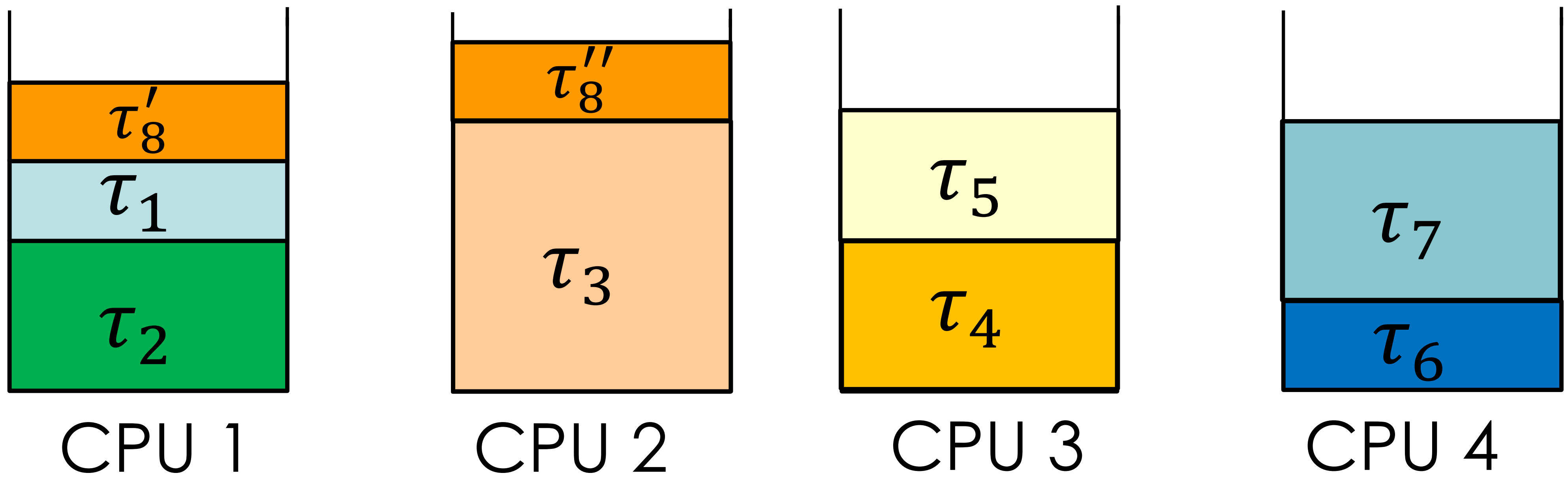
- Admission of a new reservation



1) Allocate the zero-laxity part according to the previous rule

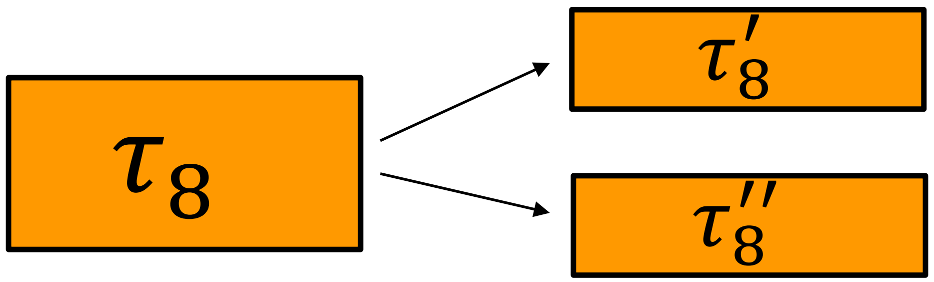
2) Allocate the remaining part using a bin-packing heuristics

$$O(m * n^{MAX})$$



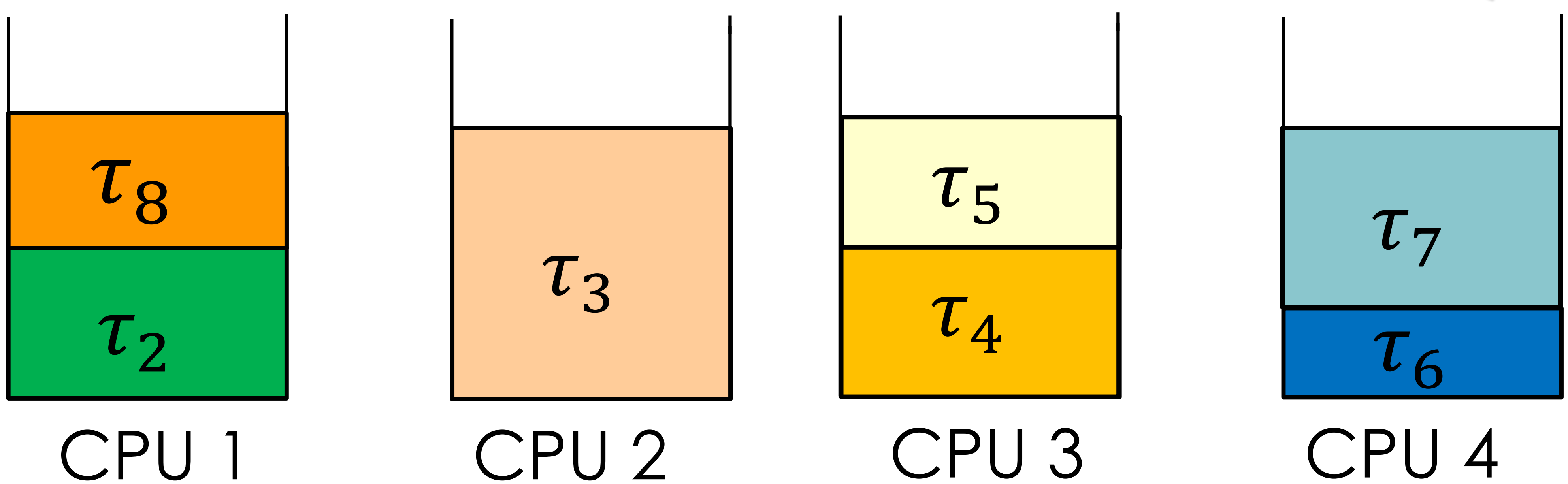
Proposed approach

Exit of a reservation



Try to recompact split reservations to favor the admission of future workload

$O(n^{MAX})$



Recall: a property of C=D Scheduling is that there can be at most m split tasks

Extensions

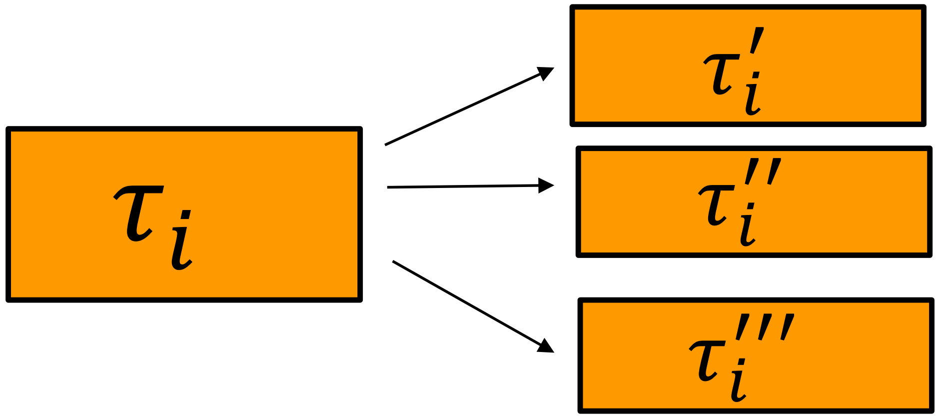
- **TAS** (Try all possible splits)

Try all possible combinations of allocations to favor the admission via splitting $O(m^2 * n^{MAX})$

- **MS** (Multi-split)

Split into multiple parts (>2)

$O(m * n^{MAX})$



- **RPR** (Reallocate Partitioned Reservation)

Move *at most one* reservation to favor the admission of a new one

$O(m^2 * n^{MAX})$

Experiments

- Sequences of events have been generated to simulate the arrival of dynamic workload

$$Event = \boxed{\{ARRIVAL, EXIT\}} \boxed{reservation}$$

- Tested generation scenarios that stress the system with high load demand

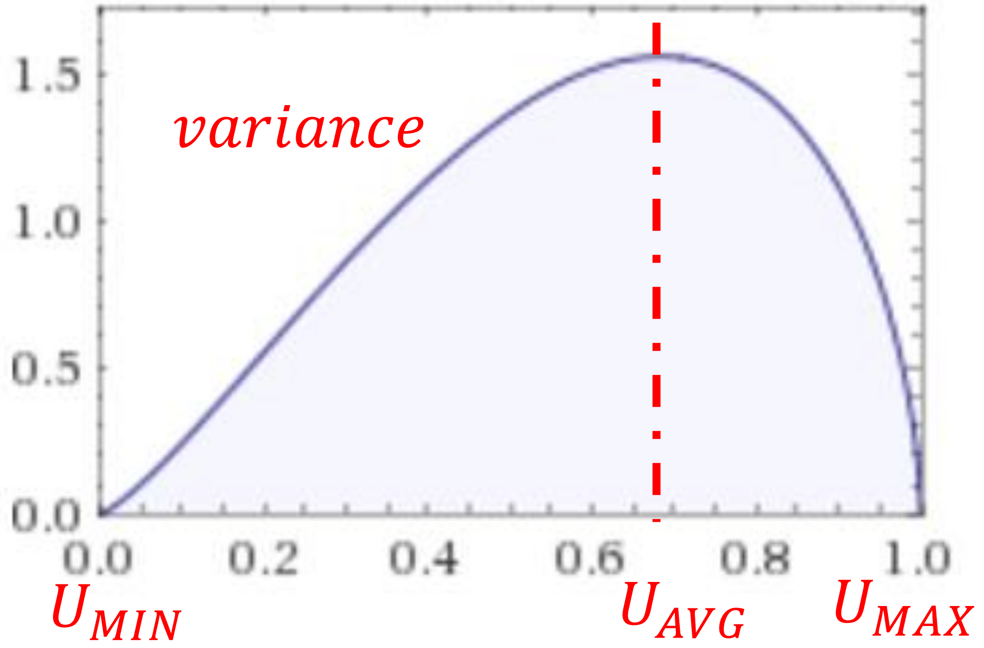
- For each generated sequence, the average accepted utilization of the proposed approach has been compared with G-EDF and P-EDF
  - G-EDF admission test is performed by combining **4** polynomial-time tests (GFB, BAK, LOAD and I-BCL)

Experiments

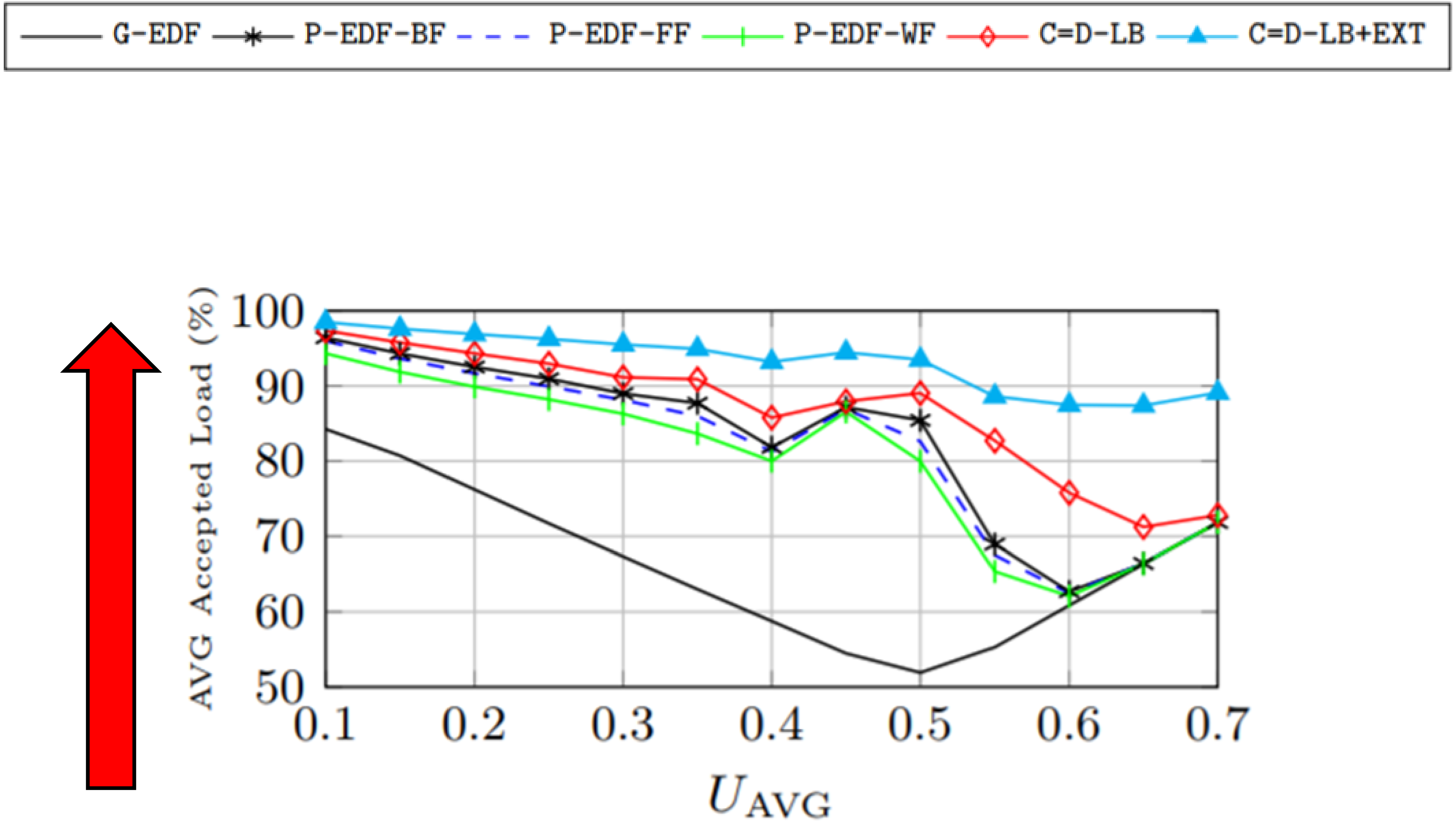
- Performance of multiprocessor scheduling algorithms are typically very sensitive to individual task utilizations

- To control average and variance of individual utilizations, reservations have been generated using the beta distribution

- Some generation parameters:
  - $[U_{\text{MIN}}, U_{\text{MAX}}] = [0.01, 0.9]$
  - $U_{AVG} \in [0.1, 0.7]$
  - $\sigma \in [0.05, 0.50]$
  - $\text{m} \in \{4, 8, 16, 32\}$



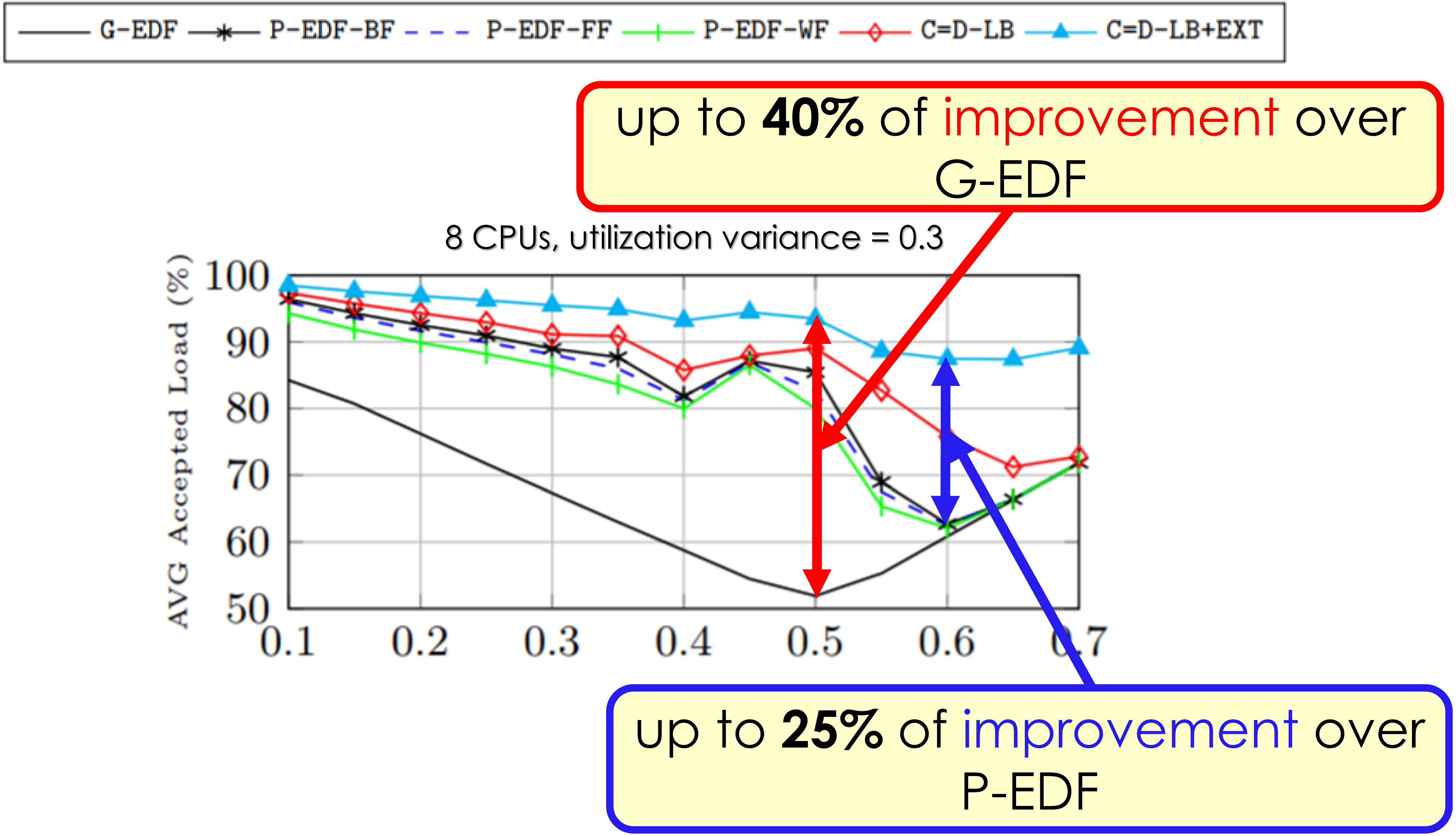
Experiments



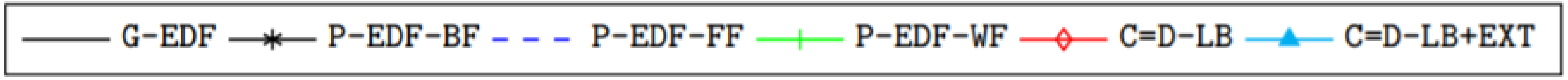
The higher the better

Increasing average task utilization

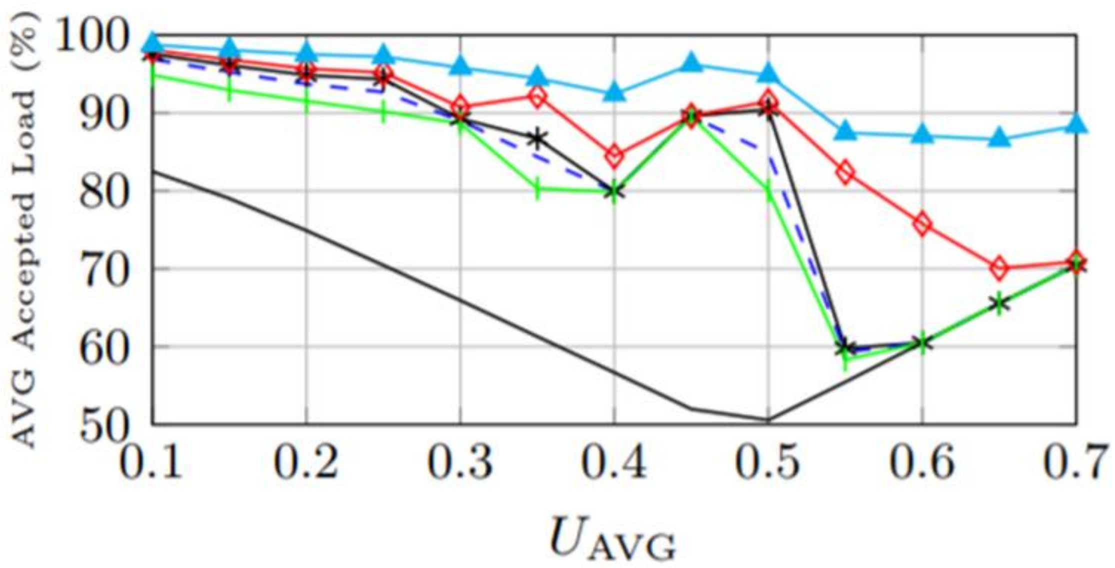
Experiments



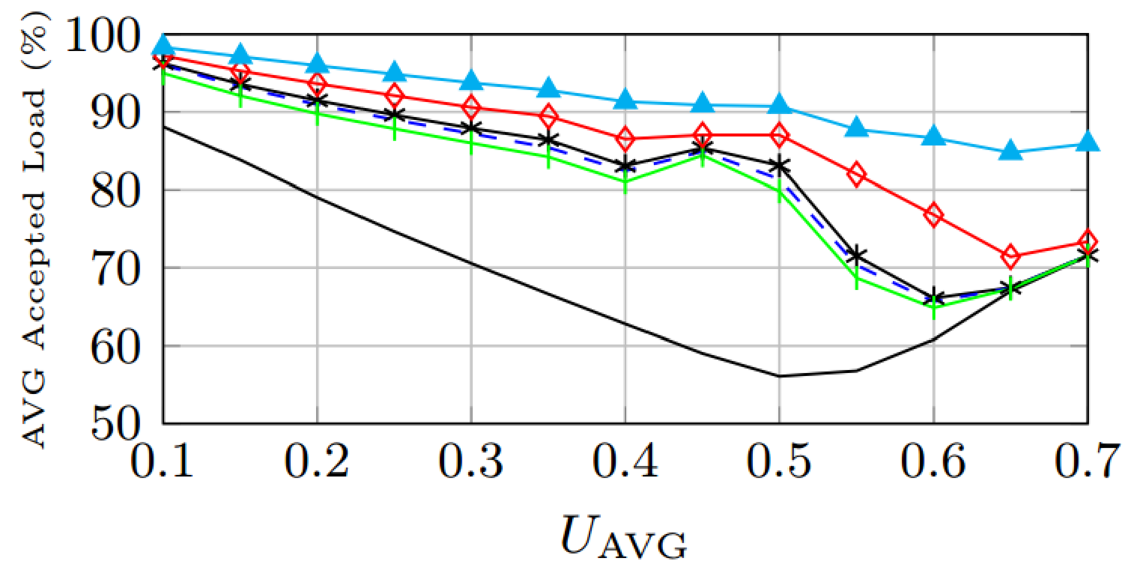
Experiments



32 CPUs, utilization variance =0.1



4 CPUs, utilization variance =0.5

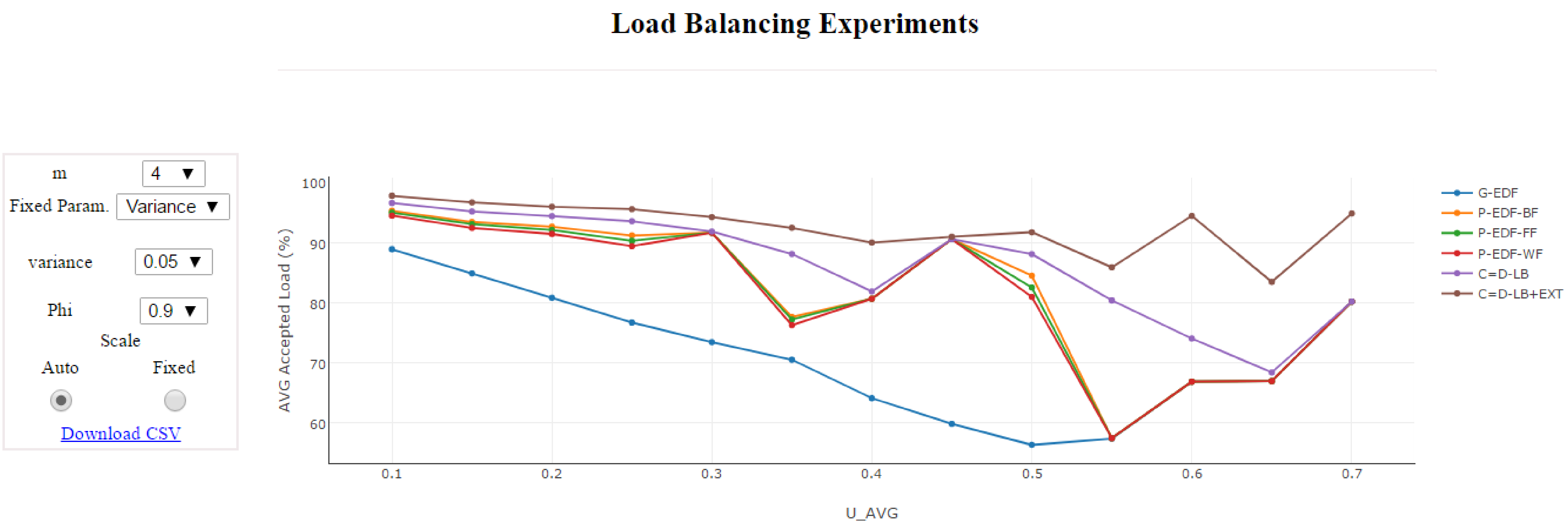


Similar trends have been observed by varying other parameters

Additional Graphs

Full set of results is freely available on-line

retis.sssup.it/~d.casini/sp-dyn/



Graphs are available for both for Load Balancing and C=D Approximation experiments

Conclusions

- We proposed a linear-time method for computing an approximation of the C=D splitting algorithm
- The approximation algorithm has been used to develop load-balacing mechanisms
- Two large-scale experimental studies have been conducted:
  - The splitting algorithm showed an average utilization loss < 3%
  - The Load Balancing mechanisms allow keeping the system load >87% with improvements up to 40% over G-EDF and up to 25% to P-EDF

Future Work

- Finding better heuristics for load balancing
- Ad-hoc mechanism for handling scheduling transients
- Support for elastic reservation to favor the admission of new workload
- Synchronization issues
- Implementation in a real-time operating systems (e.g., Linux under SCHED_DEADLINE)

Thank you!

Daniel Casini
daniel.casini@sssup.it