Technische Universität Dresden

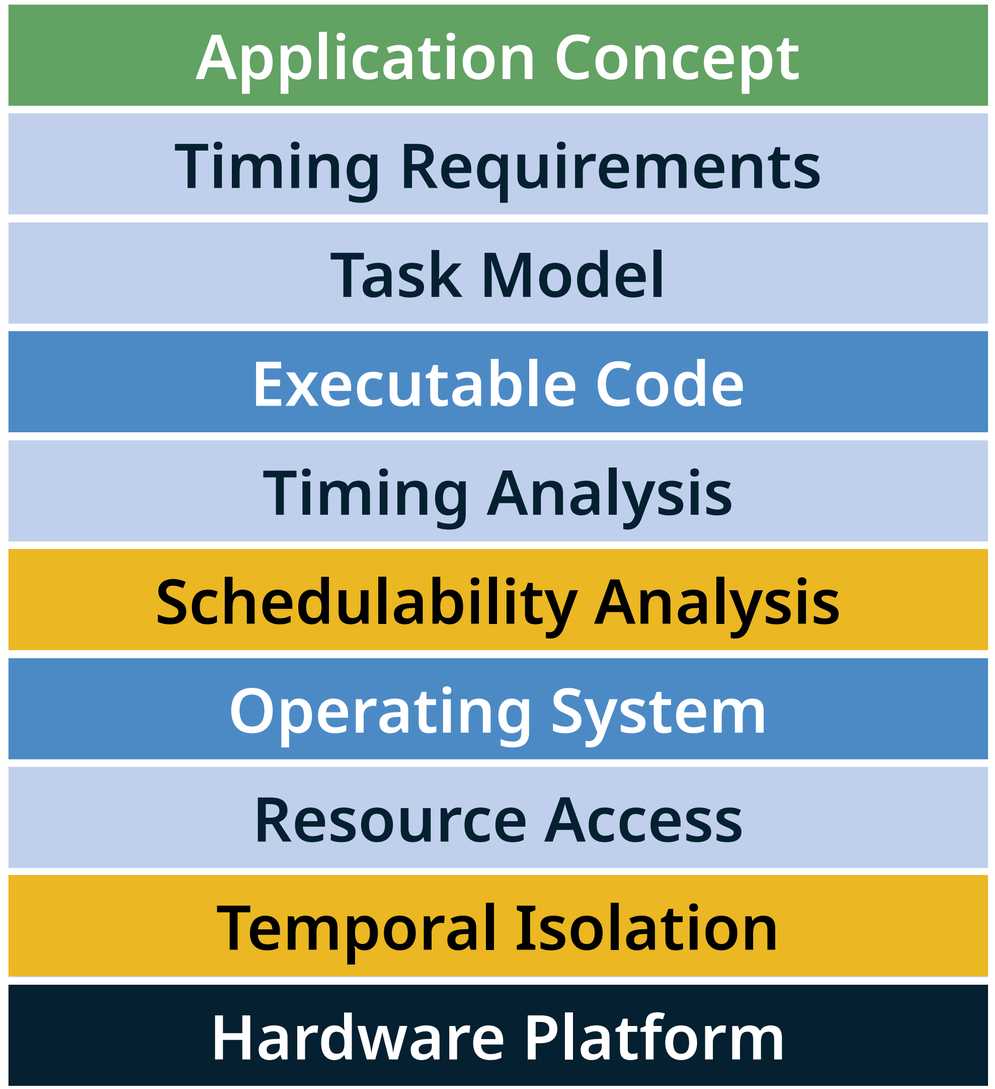
barkhausen institut

Multiprocessor Scheduling

Real-Time Systems

Michael Roitzsch

Real-Time Systems Technology Stack



Outline

- Hardware costs
- Anomalies & impossibility results
- Partitioned multiprocessor scheduling (no migration)
- Global multiprocessor scheduling (task- / job-level migration)
- Optimal multiprocessor scheduling

Lessons Learned From Uniprocessor Scheduling

- Liu-Layland utilization bound for fixed task priority algorithms
- fixed job priority algorithms are optimal: EDF
- there are optimal greedy algorithms: a single measure characterizes the ‘importance’ of a job (e.g., time to deadline, laxity, …)
- all preemptive FTP, FJP algorithms are predictable
- all preemptive FTP algorithms and EDF are sustainable
- simultaneous release is the critical instant
- response times depend on the *set* but not on *order* of high-priority tasks

Overview of Multiprocessor Scheduling

Two problems to solve:

- Priority Problem: When to run a given job of the workload?
  - fixed task priority (e.g., RMS, ...)
  - fixed job priority (e.g., EDF, ...)
  - dynamic job priority (e.g., LST, ...)
- Allocation Problem: Where to run this job?
  - no migration
  - task-level migration (no migration of running jobs)
  - job-level migration (migrate also running jobs)

Overview of Multiprocessor Scheduling

Two problems to solve:

- Priority Problem: When to run a given job of the workload?
  - fixed task priority (e.g., RMS, ...)
  - fixed job priority (e.g., EDF, ...)
  - dynamic job priority (e.g., LST, ...)

- Allocation Problem: Where to run this job?
  - no migration
  - task migration (migrate also running jobs)
  - job-level migration (migrate also running jobs)

Preemption Costs – UP & MP

Migration Costs – MP

Preemption Costs

Direct Costs

- timer / device interrupt
- save register state
- manipulate ready list
    - uniprocessor: no synchronization required
- load register state of next thread

Indirect Costs

- cache evictions between two consecutive runs
- TLB refills after evictions / flush

Migration Costs

Job-level Migration

- migration of running job implies preemption at source CPU

Task-level Migration

- job is already preempted

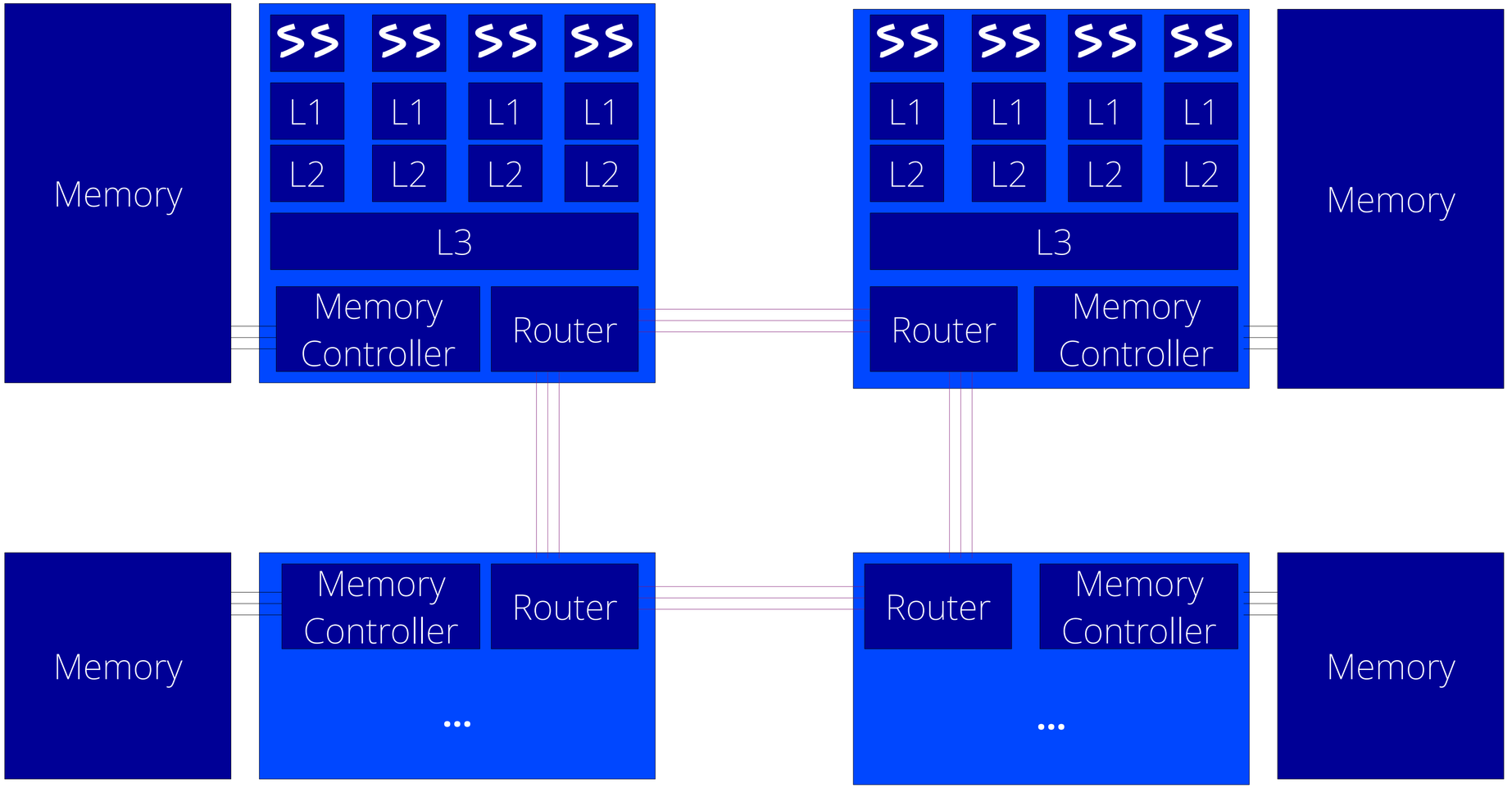
Direct Costs

- manipulate remote / global ready list (synchronization!)
- fetch register state

Indirect Costs

- populate TLB entries, TLB shootdown costs
- fetch active cache working set from remote cache
- load remaining data from remote memory (NUMA!)

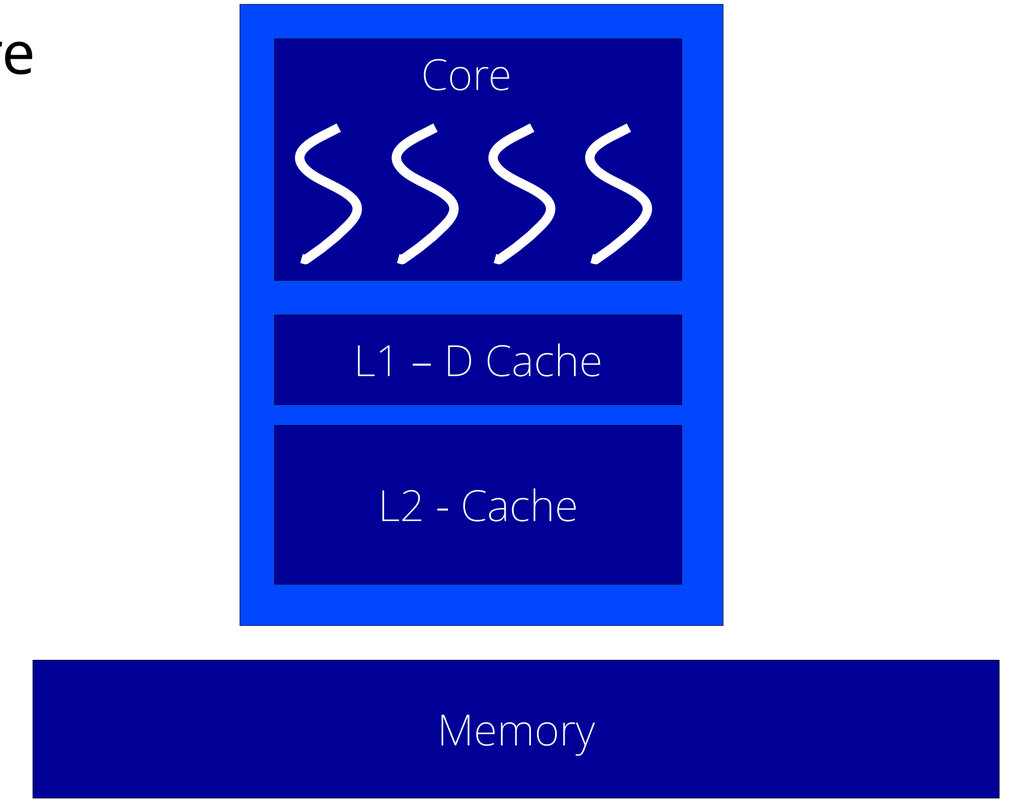
Multiprocessor Architectures



Multiprocessor Architectures

Symmetric Multi-Threading (SMT)

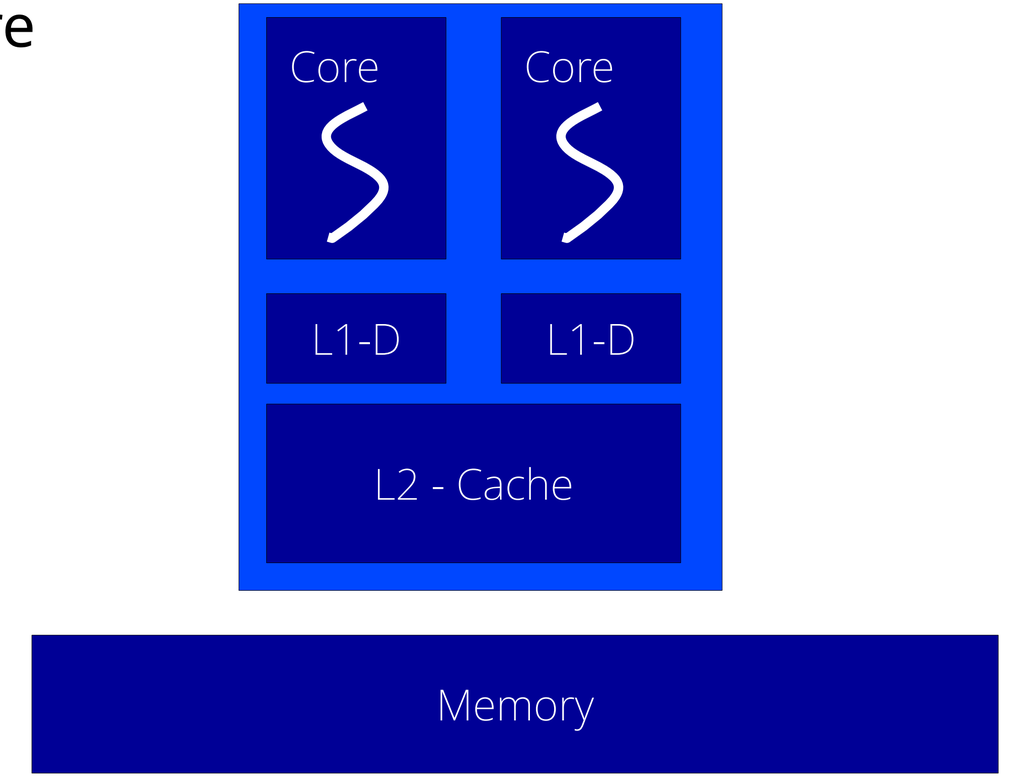
- operating system multiplexes n software threads on m hardware threads
- caches + pipeline + ALUs are shared
- no indirect migration costs
- performance predictability difficult: depends on microarchitecture



Multiprocessor Architectures

Multi-Core Processors

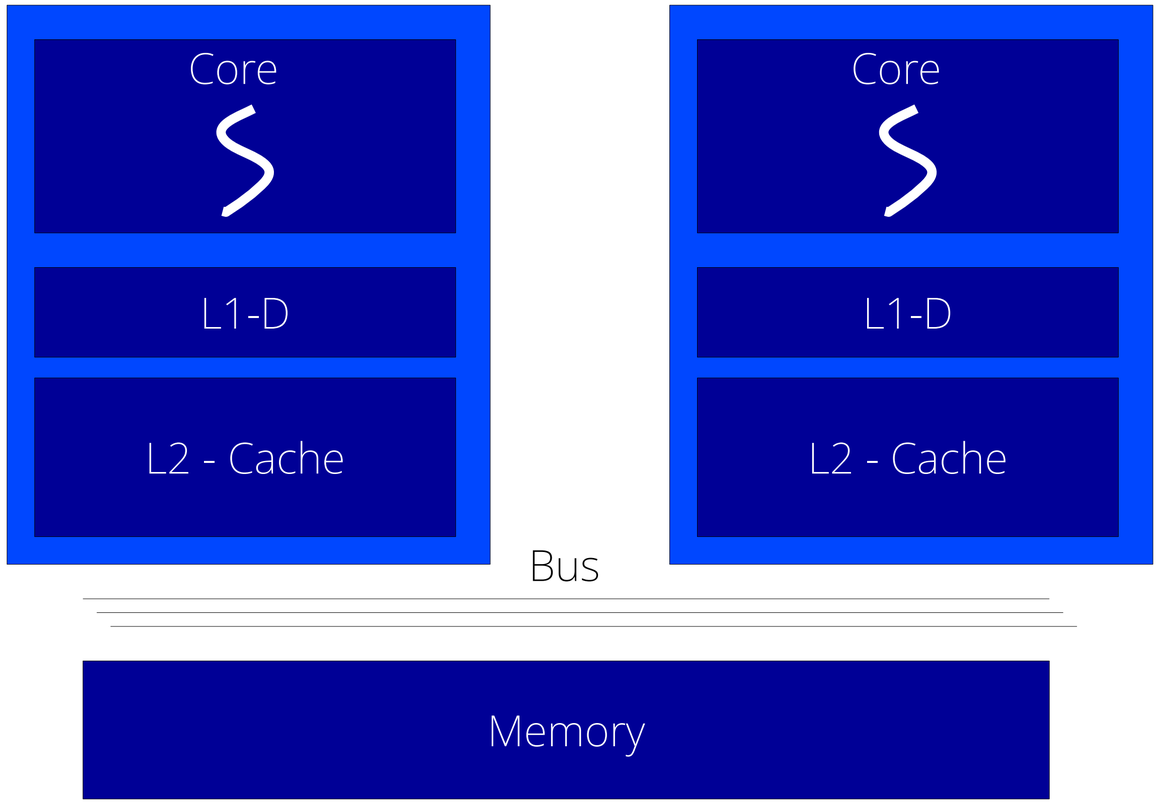
- operating system multiplexes n software threads on m cores
- timing of last level cache dominates migration costs



Multiprocessor Architectures

Symmetric Multiprocessors

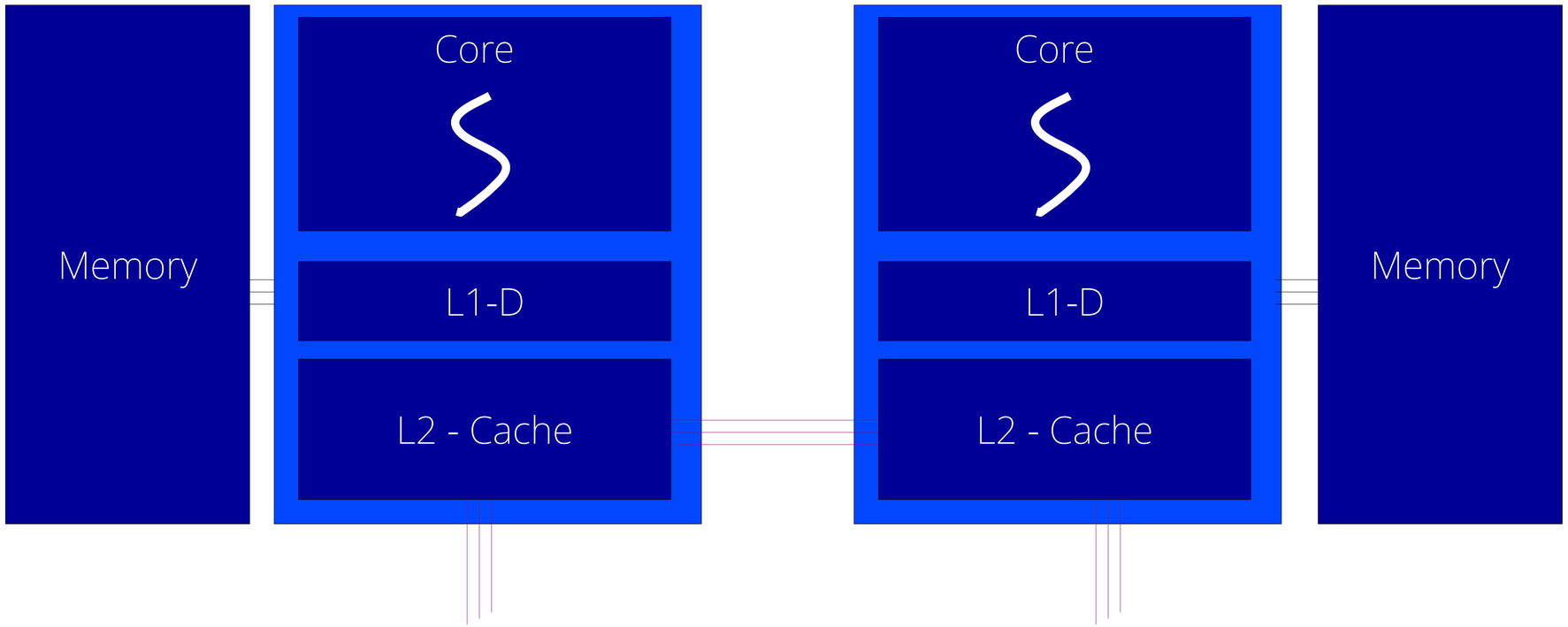
- operating system multiplexes n software threads on m dies
- timing of interconnect dominates migration costs



Multiprocessor Architectures

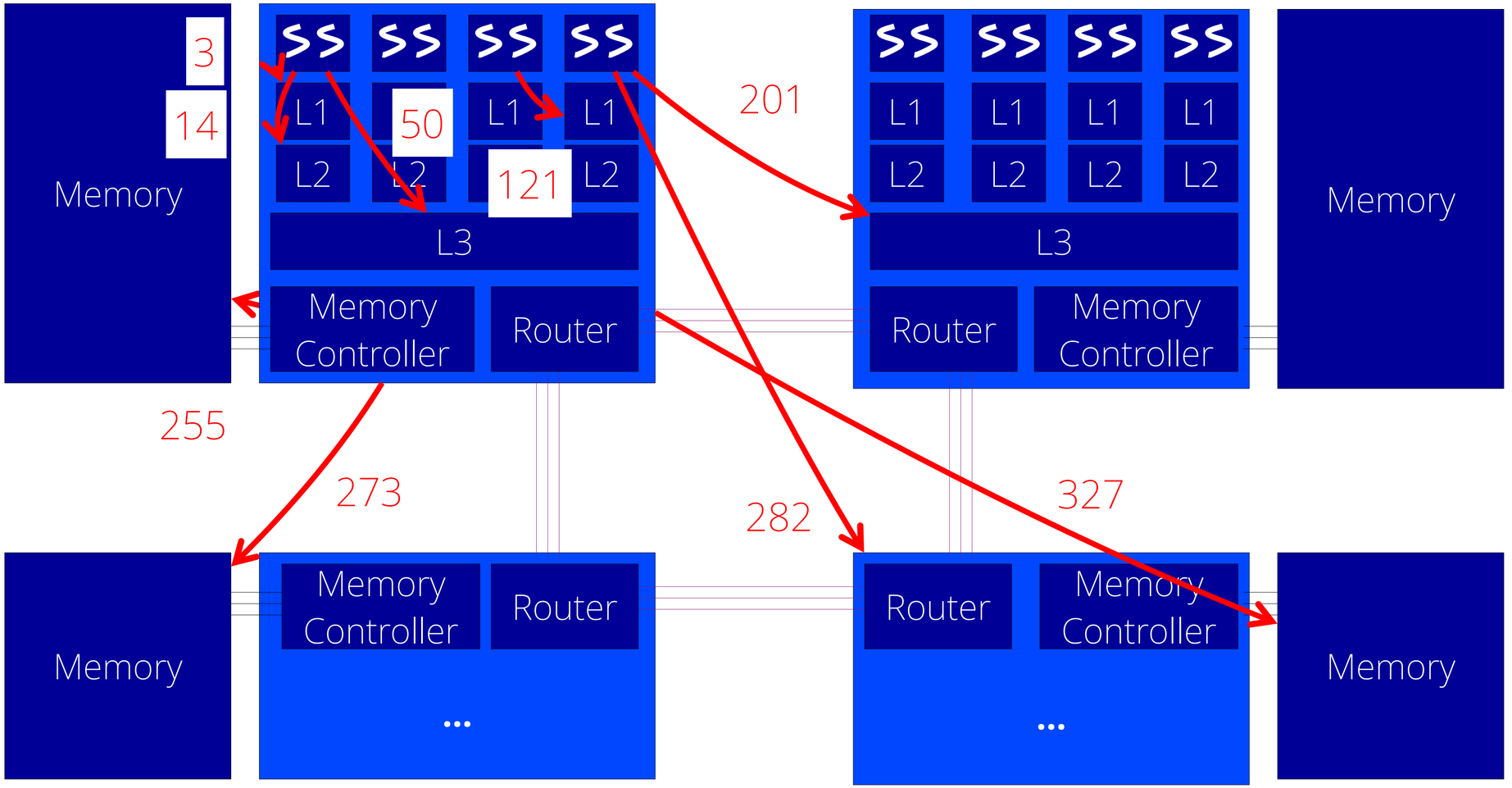
Cache-Coherent NUMA

non-uniform memory access: fetches may go to remote memory



Multiprocessor Architectures

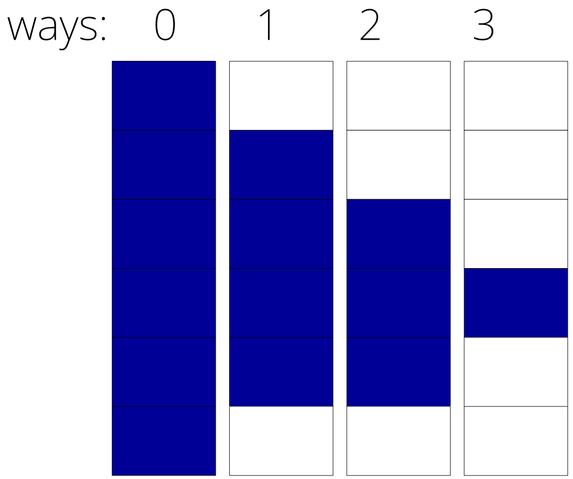
Corey – OSDI '08



Migration Costs

Active Cache Working Set

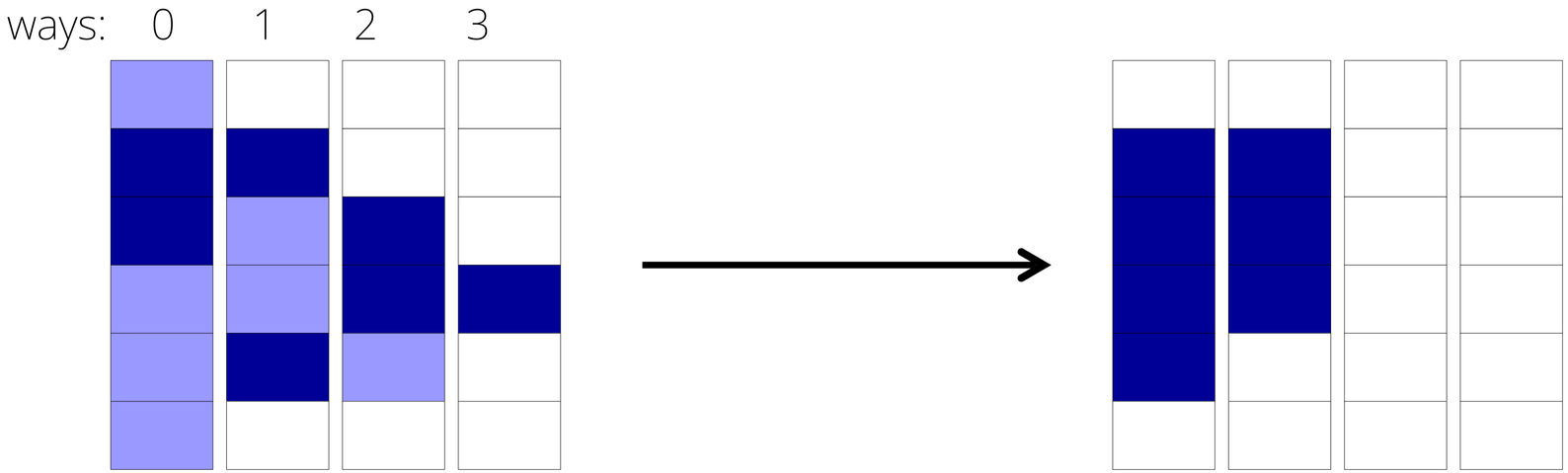
- cachelines a thread would access again if it would run
- varies over time
- ages out after preemption



Migration Costs

Active Cache Working Set

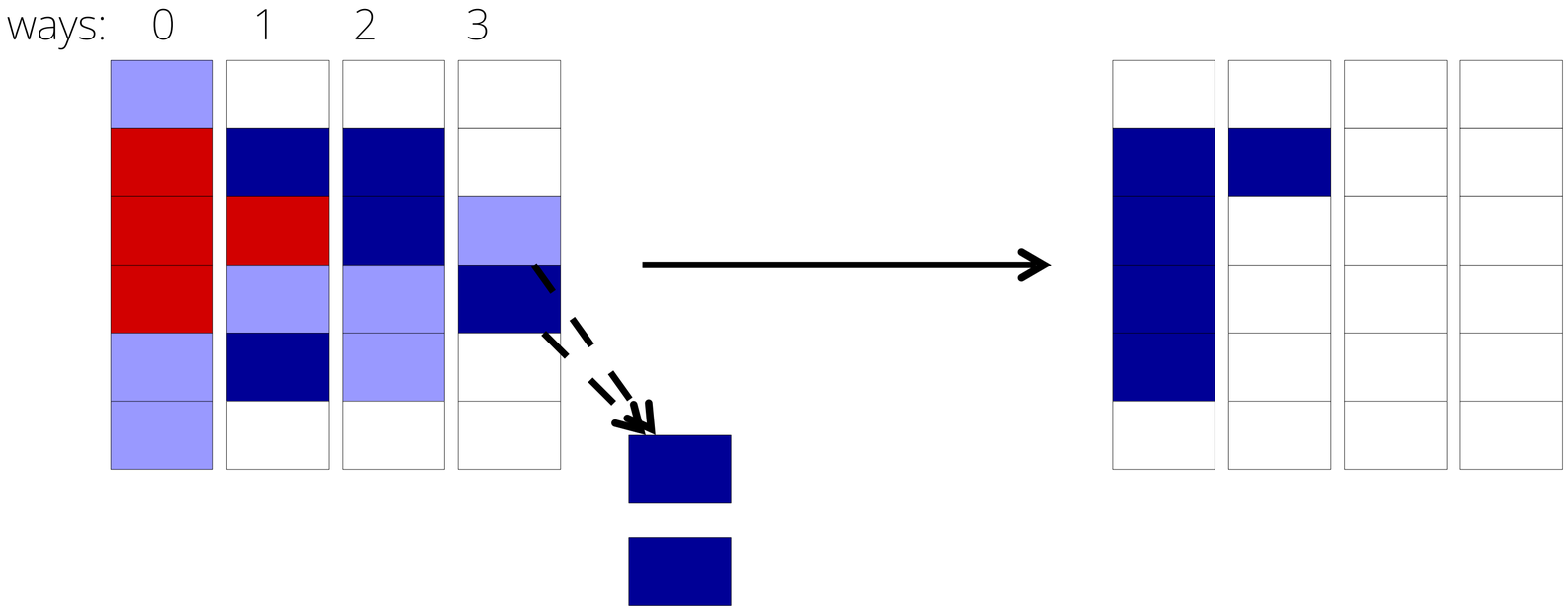
- cachelines a thread would access again if it would run
- varies over time
- ages out after preemption



Migration Costs

Active Cache Working Set

- cachelines a thread would access again if it would run
- varies over time
- ages out after preemption



Migration Costs

Summary

- migration costs are highly architecture dependent
- non-trivial to predict
- may cause a significant delay when a thread resumes execution

Assumption for the remainder of this lecture:

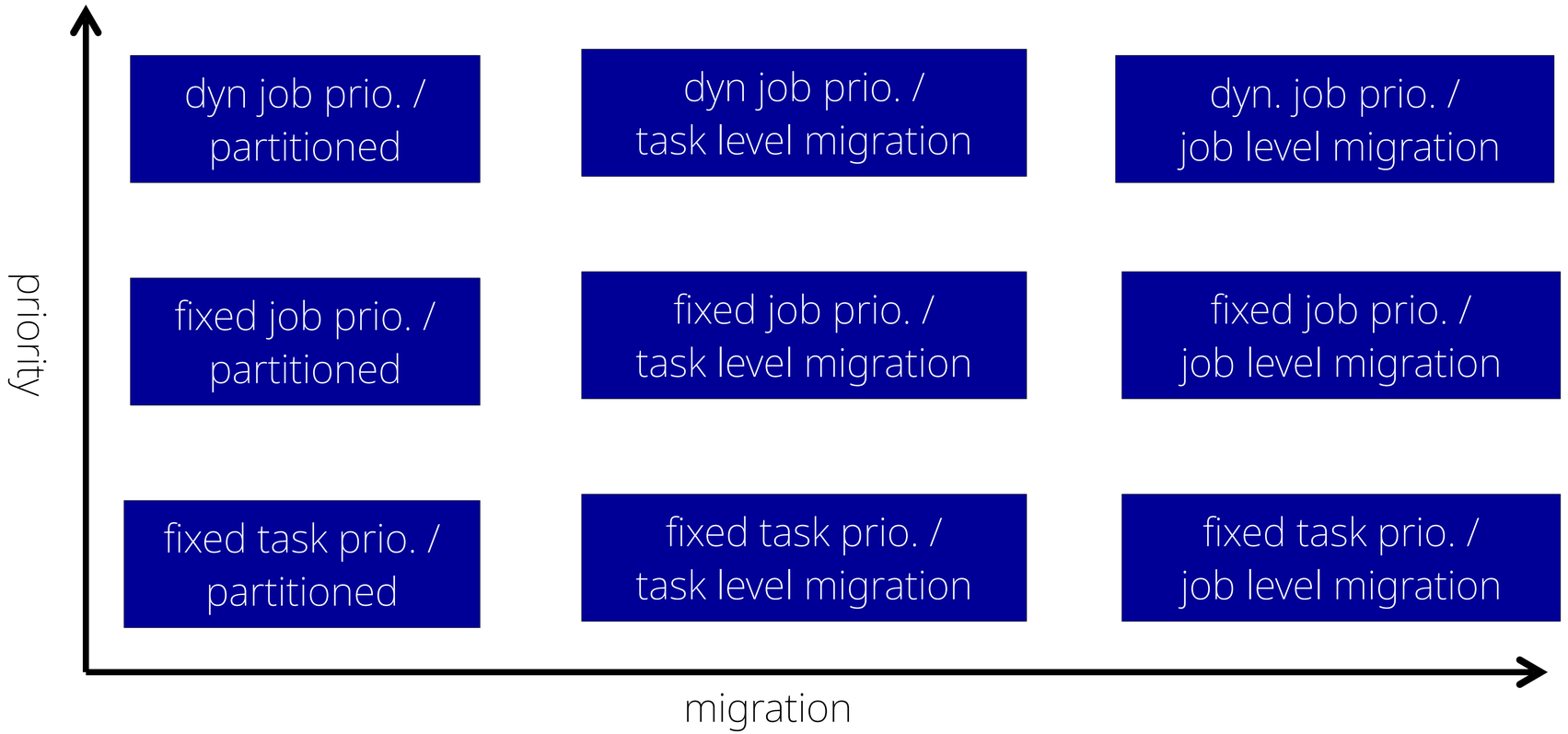
preemption and migration costs attributed to WCET

Technische Universität Dresden

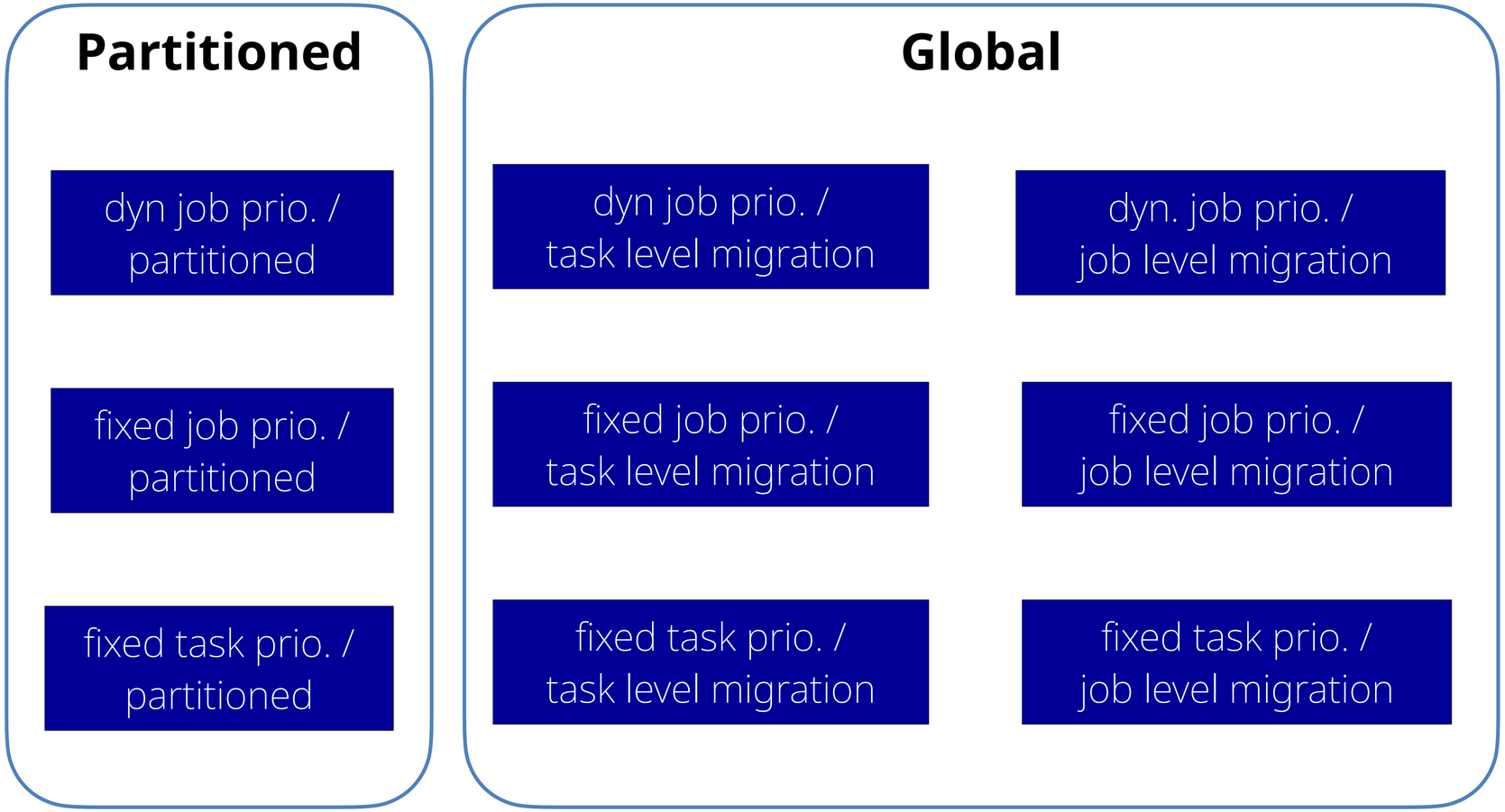
barkhausen institut

Anomalies & Impossibilities

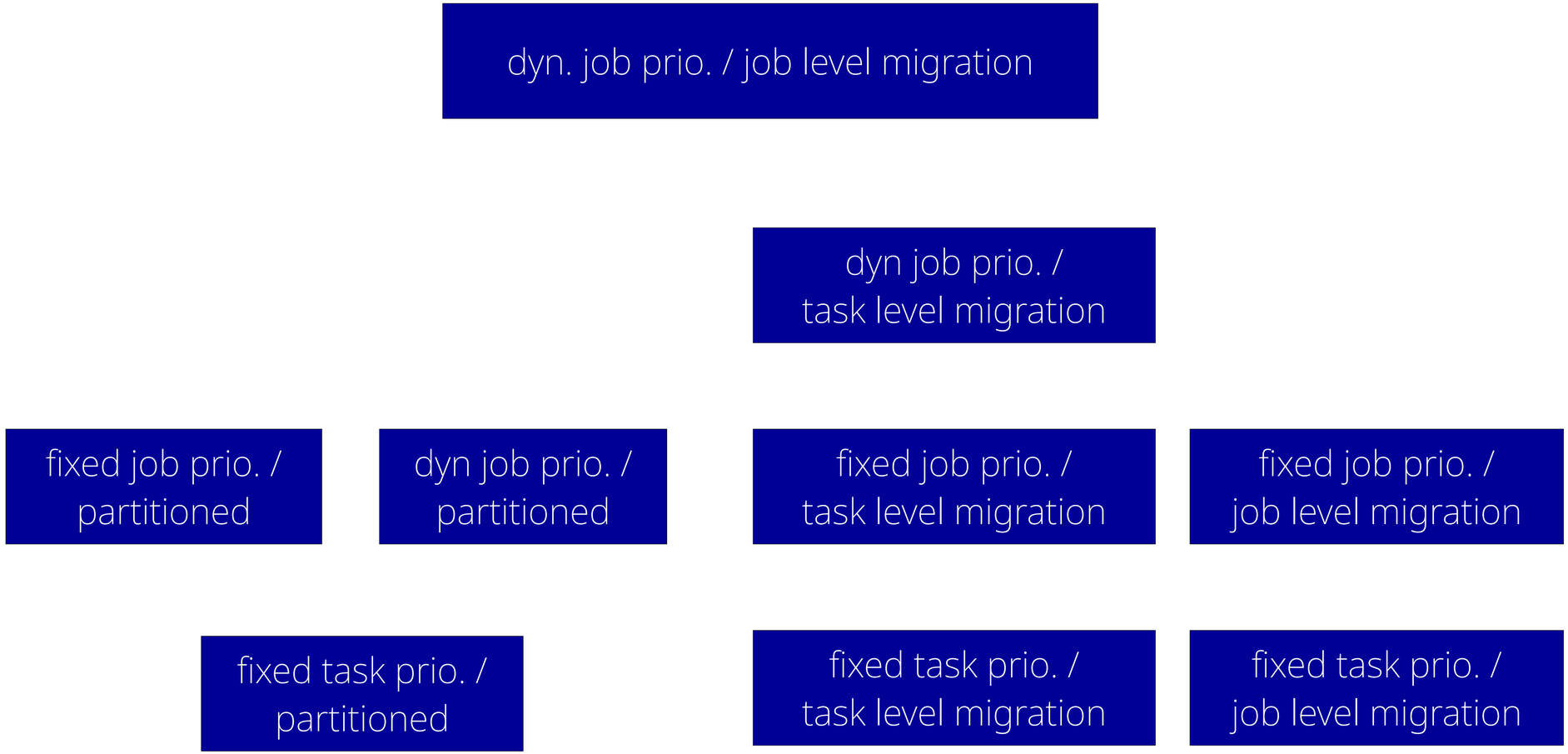
Design Space of Multiprocessor Scheduling



Design Space of Multiprocessor Scheduling



Design Space of Multiprocessor Scheduling



Relative ordering between classes of scheduling algorithms: later in this lecture

Terminology, Notation and Assumptions

Periodic Tasks

- Task $t_i = (P_i, D_i, C_i)$ P_i = const.

Sporadic Tasks

- P_i = minimum inter-release separation

Deadlines

- implicit deadline: $D_i = P_i$ (relative deadline = period)
- constrained: $D_i \leq P_i$ (relative deadline < period)
- arbitrary (deadline may be after period end)

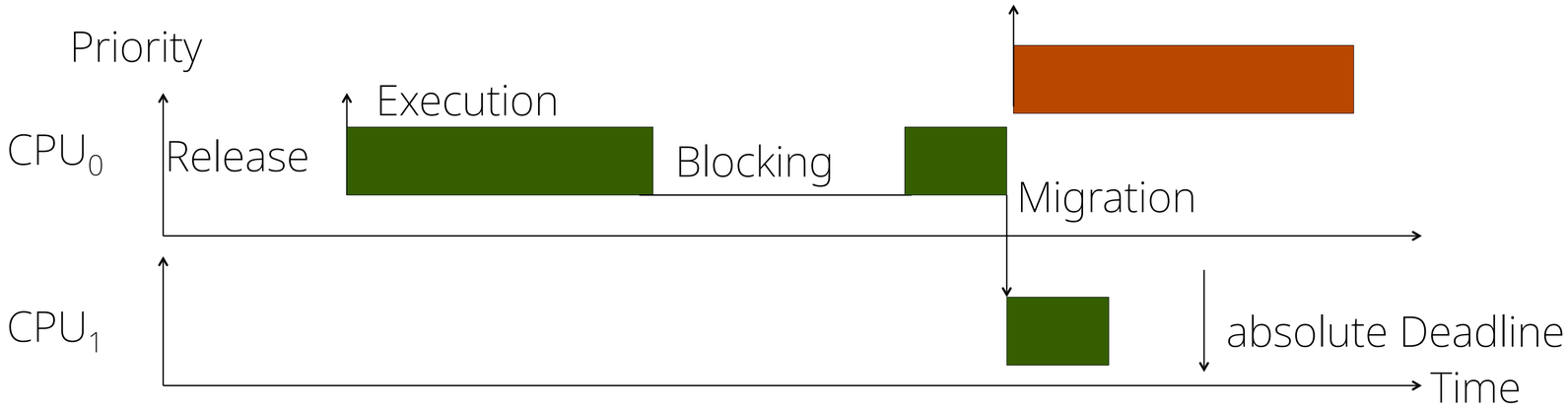
Utilization $U_i = \frac{C_i}{P_i}$

Density $\partial_i = \frac{C_i}{min(D_i, P_i)}$

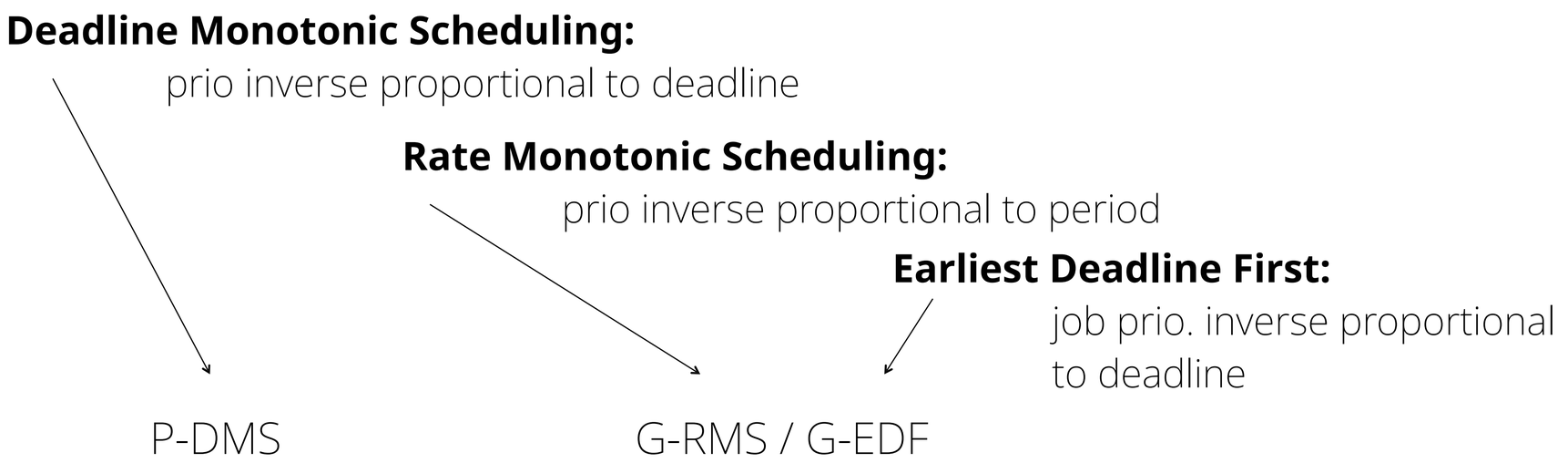
Terminology, Notation and Assumptions

Assumptions for the remainder of this lecture

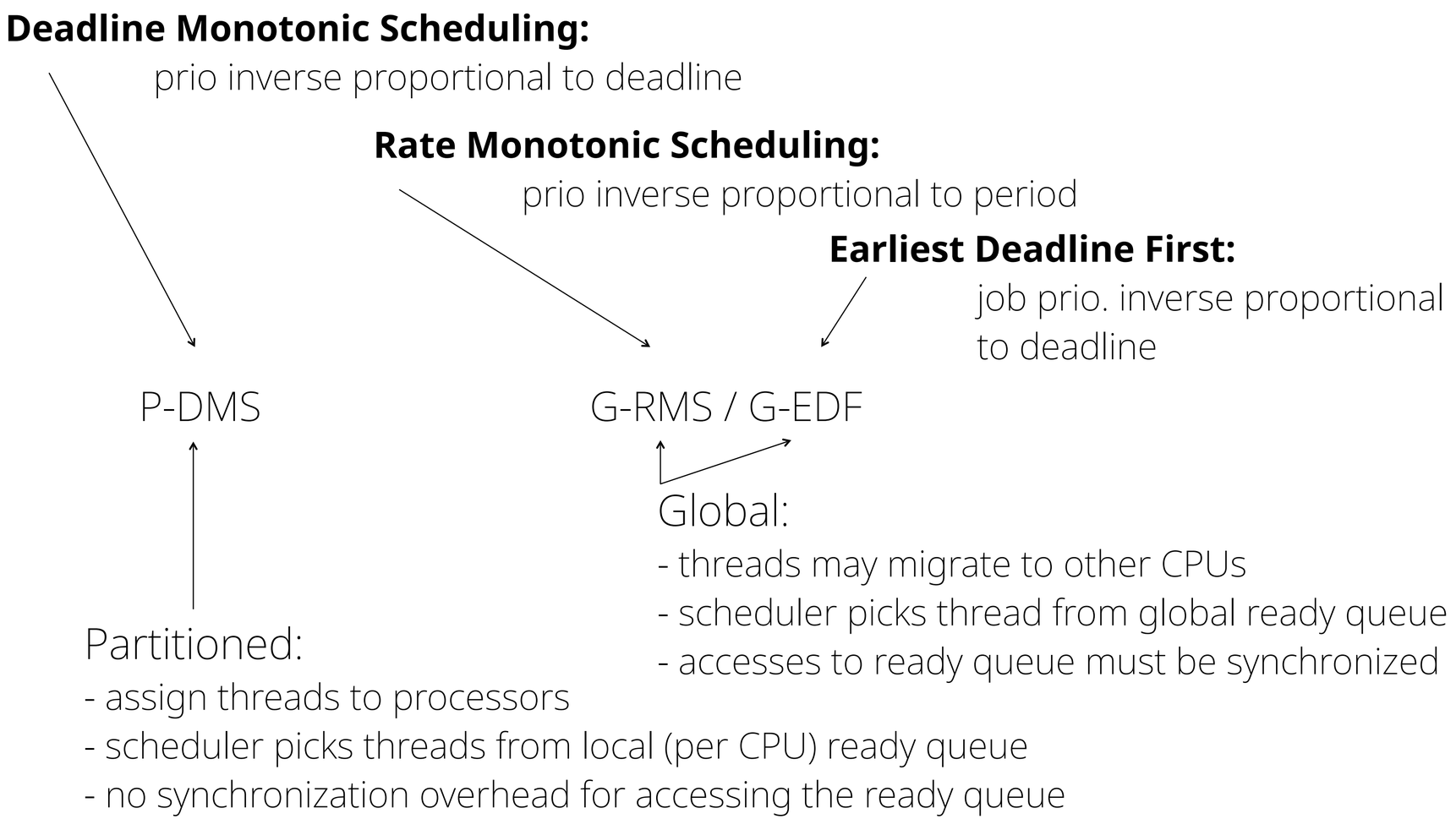
- independent tasks, jobs do not block
- fully preemptible / migratable (costs in WCET)
- unlimited number of priorities
- tasks are single threaded: a job can utilize only 1 CPU at a time



Terminology: P-DMS / G-RMS / G-EDF



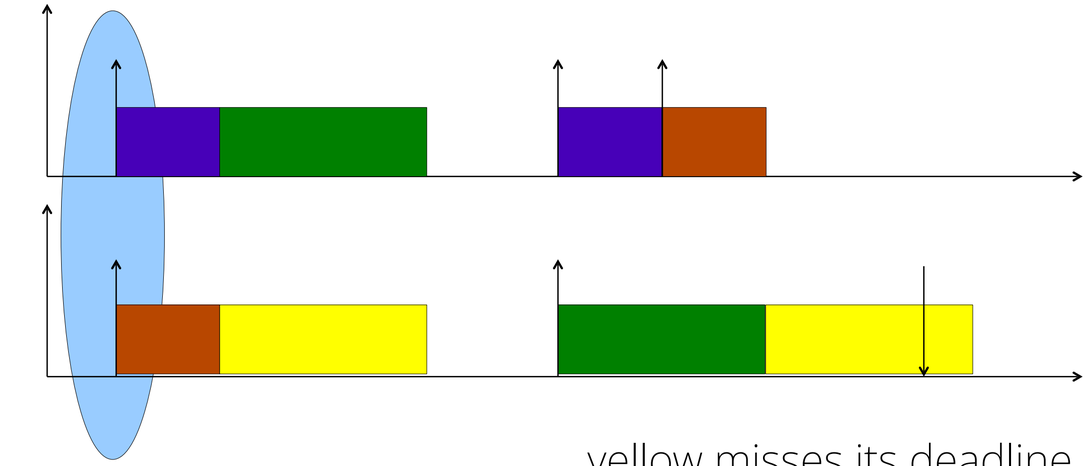
Terminology: P-DMS / G-RMS / G-EDF



Anomalies

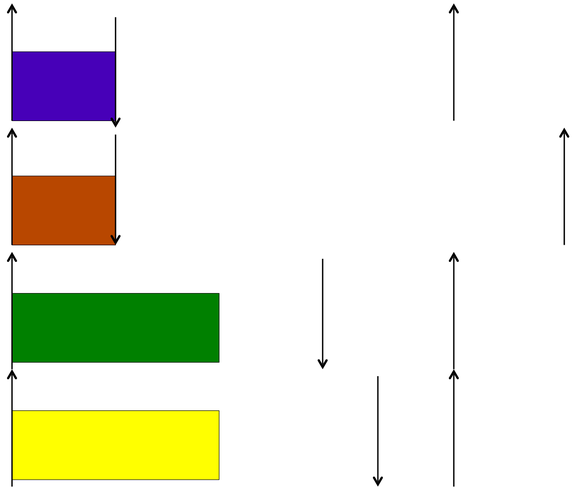
Simultaneous Release is not Critical Instance — *Lauzac '98*

longer response time in second period



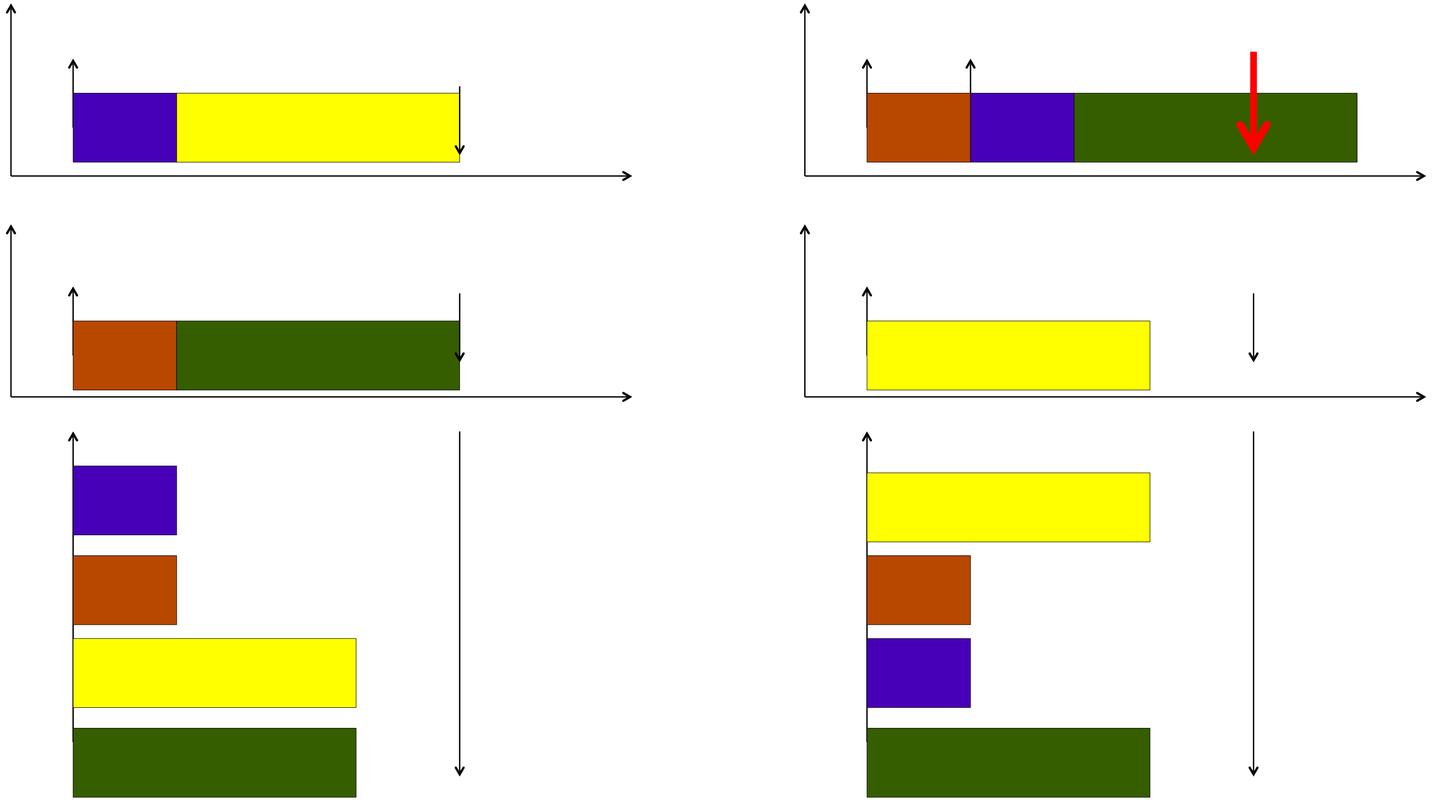
yellow misses its deadline

simultaneous release of all tasks



Anomalies

Response time (of green) depends not only on set of higher prioritized tasks but also on their relative priority ordering



Sustainability

A schedulable workload remains schedulable if we

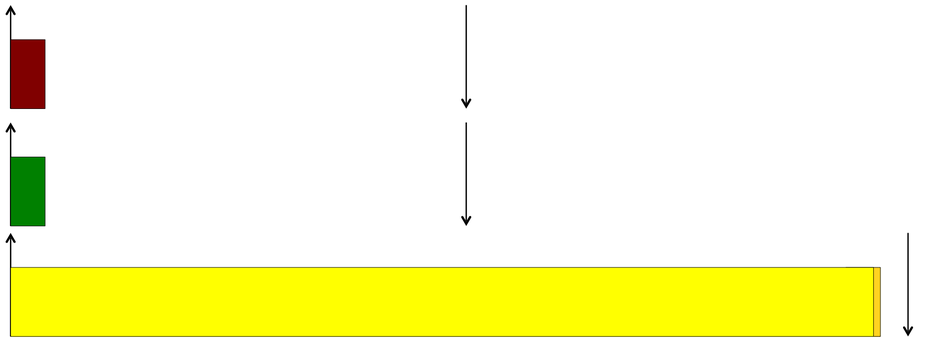
- decrease the execution time of a task
  otherwise, WCET won't work as admission criterion
- increase the minimal interrelease time (period) of a task
  otherwise, more frequent recurrence is no safe approximation
- increase the relative deadline of a task
  otherwise, earlier deadline is no safe approximation

- G-FTP + G-EDF are not sustainable if #CPUs > 1
- all preemptive FJP / FTP algorithms are predictable

Dhall Effect

The utilization bound of Global EDF is as low as $U_{EDF} = 1 + \varepsilon$

- m tasks with short periods and infinitesimally low U_i (e.g., $U_i = \varepsilon$)
- 1 task with larger period and U_j close to 1 (e.g., $U_j > (2 - \varepsilon) / 2$)

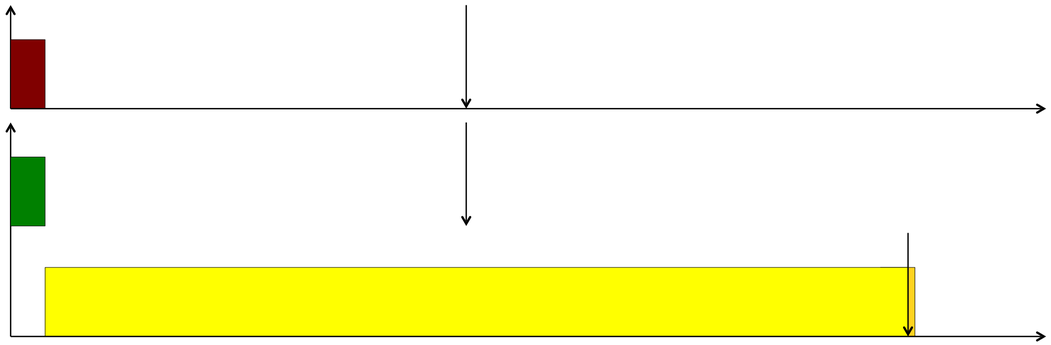


Dhall Effect does not manifest if $U_i < 41\%$

Dhall Effect

The utilization bound of Global EDF is as low as $U_{EDF} = 1 + \varepsilon$

- m tasks with short periods and infinitesimally low U_i (e.g., $U_i = \varepsilon$)
- 1 task with larger period and U_j close to 1 (e.g., $U_j > (2 - \varepsilon) / 2$)



Dhall Effect does not manifest if $U_i < 41\%$

Some Impossibility Results

Hong '88

No optimal online multiprocessor scheduling algorithm for arbitrary collections of jobs, unless all jobs have the same relative deadline.

Dertouzos '89

Even if execution times are known precisely, clairvoyance for job arrivals is necessary for optimality.

Fisher '07

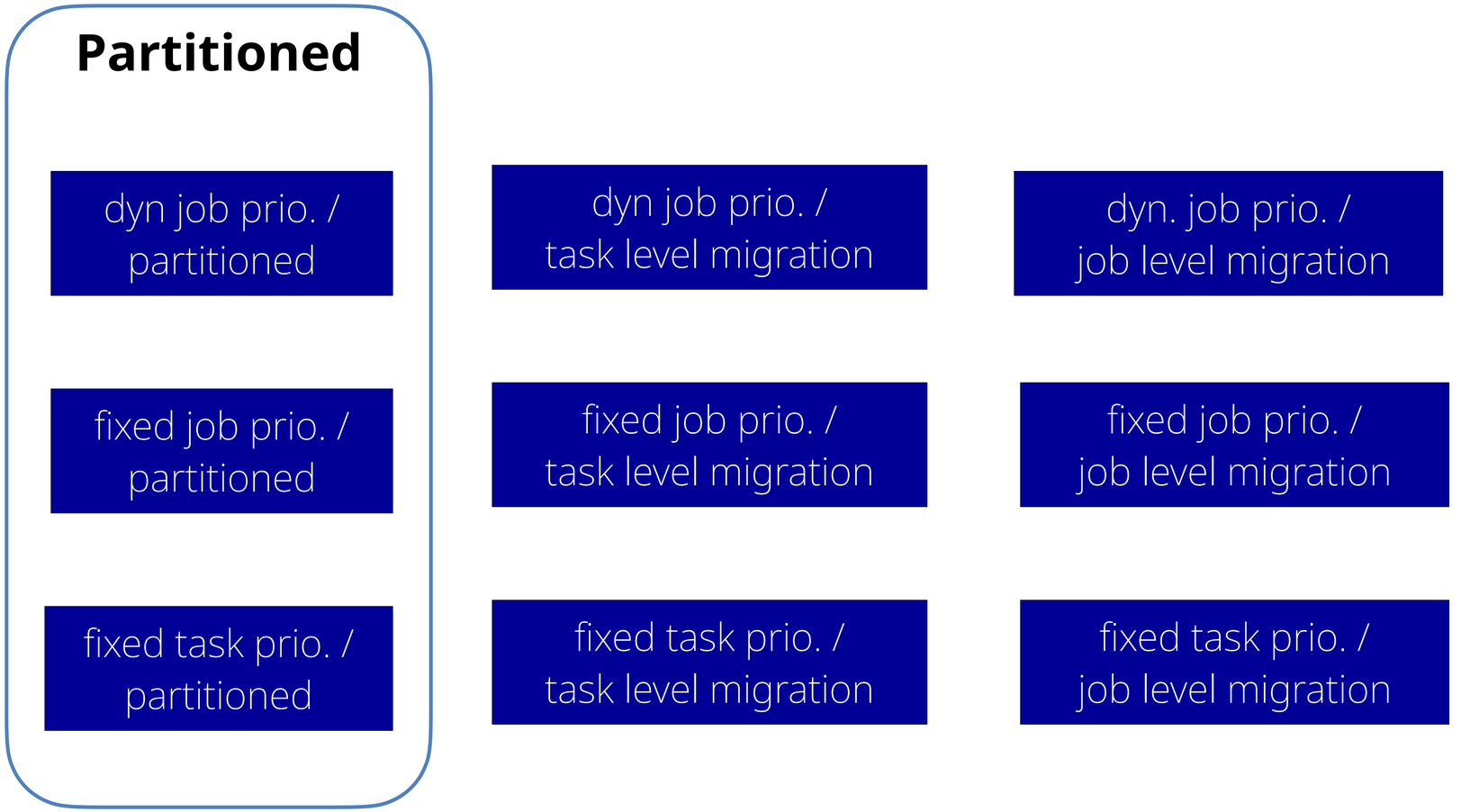
No optimal online algorithm for sporadic tasksets with constrained or arbitrary deadlines.

Technische Universität Dresden

barkhausen institut

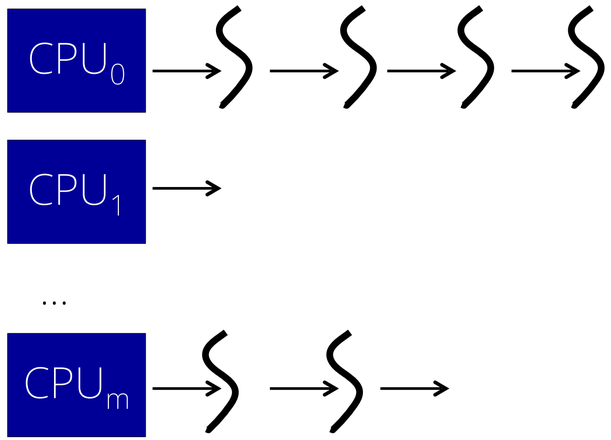
Scheduling Algorithms

Partitioned Scheduling



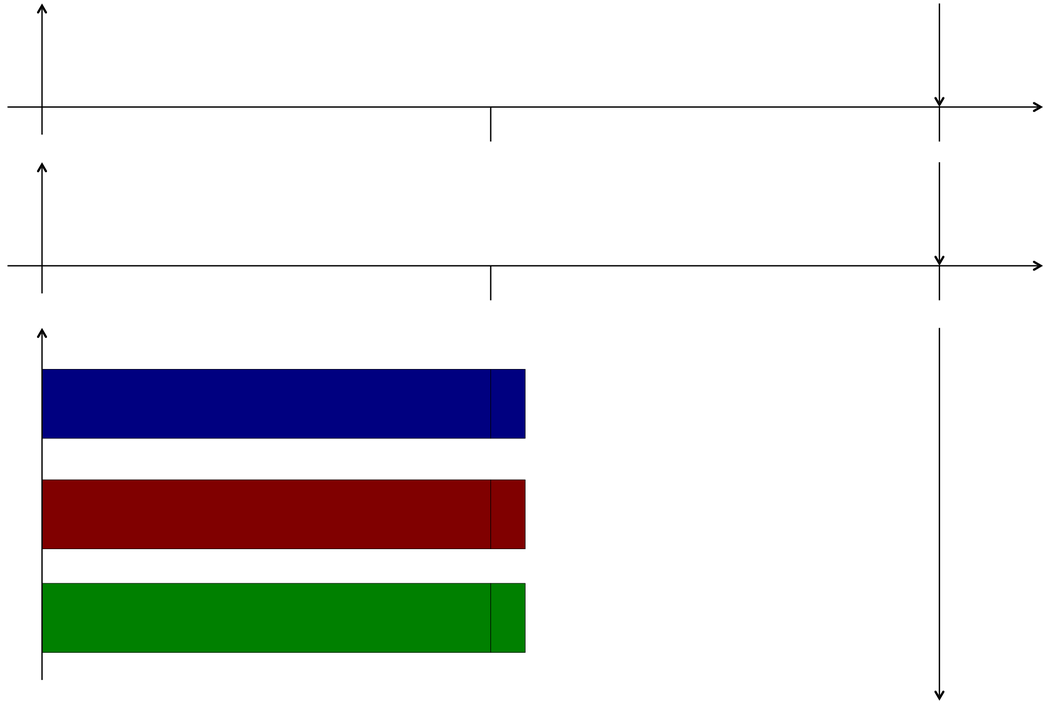
Partitioned Scheduling

- split workload by allocating tasks to CPUs
- run allocated task with uniprocessor scheduling algorithm
- reap benefit of well known uniprocessor results
- optimal task allocation is NP complete: maps to binpacking



Partitioned Scheduling

Utilization bound for implicit deadline workloads — *Anderson '01*

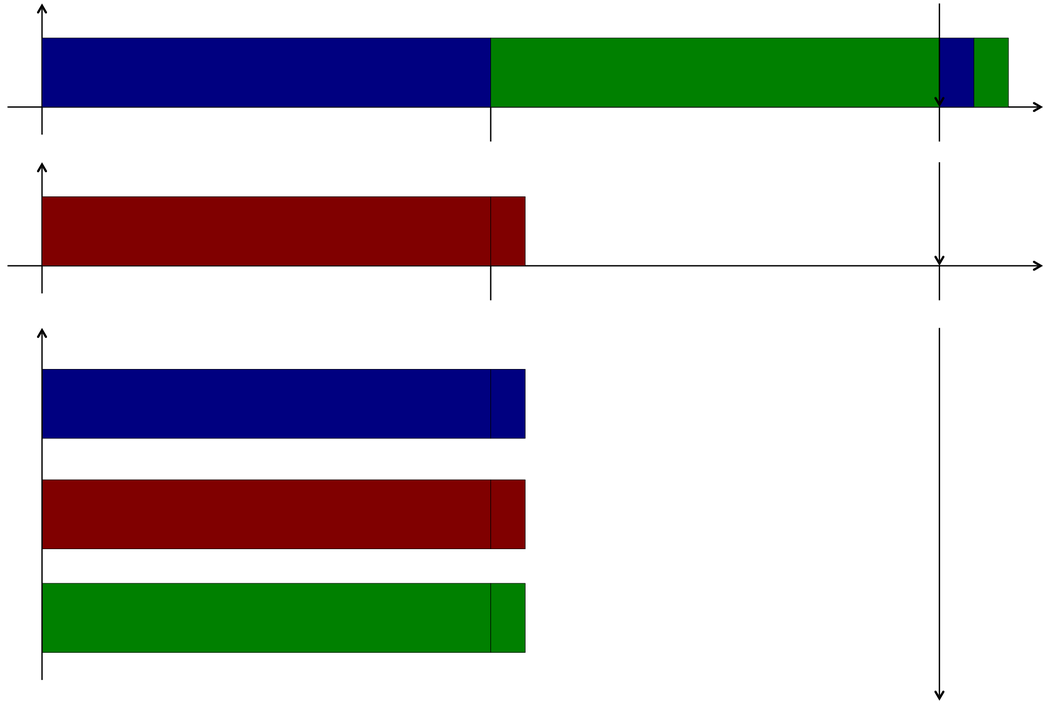


$$U_{opt} = \frac{m+1}{2}$$

No partitioning scheduling algorithm can produce a feasible schedule of m+1 tasks with execution time 1+ε and period of 2 on m processors

Partitioned Scheduling: Utilization Bound

Utilization bound for implicit deadline workloads — *Anderson '01*

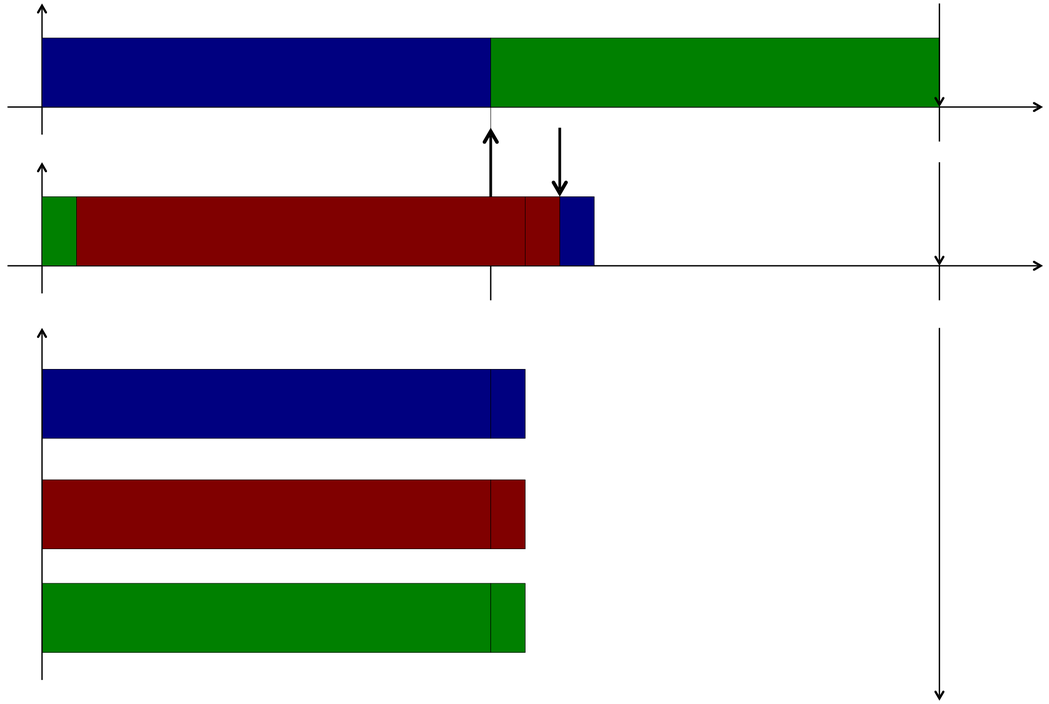


$$U_{opt} = \frac{m+1}{2}$$

No partitioning scheduling algorithm can produce a feasible schedule of m+1 tasks with execution time 1+ε and period of 2 on m processors

Partitioned Scheduling: Utilization Bound

Utilization bound for implicit deadline workloads — *Anderson '01*



$$U_{opt} = \frac{m+1}{2}$$

Easy if blue and green can migrate to CPU2.

PDMS-HPTS-DS — *Lakshmanan '09*

Can we improve on Anderson's Utilization Bound?

by allowing a few jobs to migrate

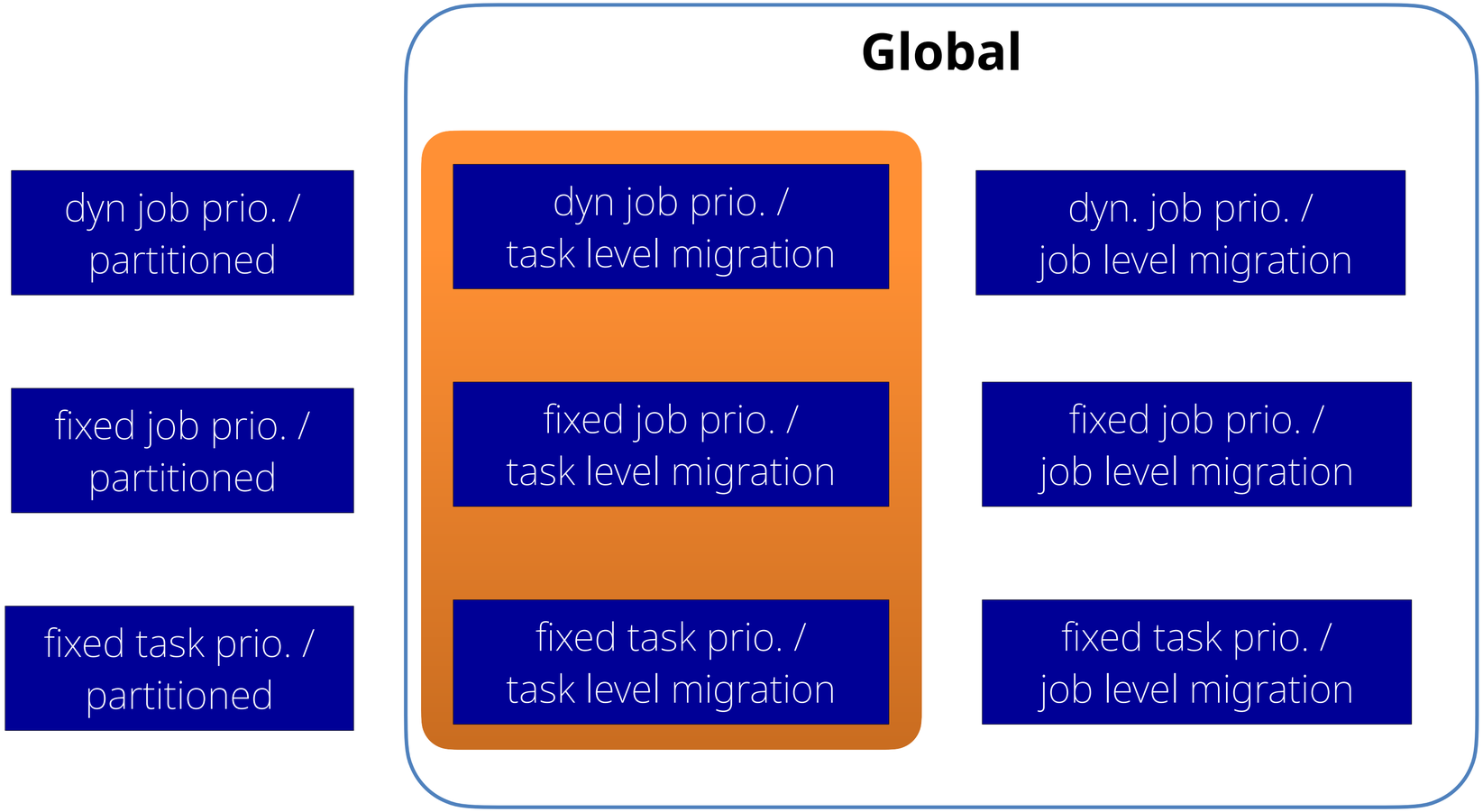
PDMS Partitioned Deadline Monotonic Scheduling

HPTS Split Highest Priority Task

DS Allocate according to Highest Density First

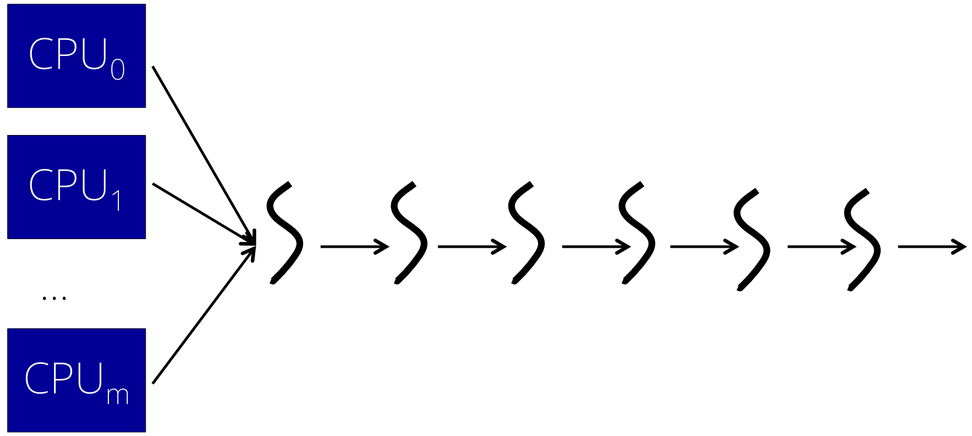
$U_{\text{PDMS-HPTS-DS}} = 69.3\ \%$ if all tasks have a utilization $U_i < 41\%$

Global Scheduling



Global Scheduling

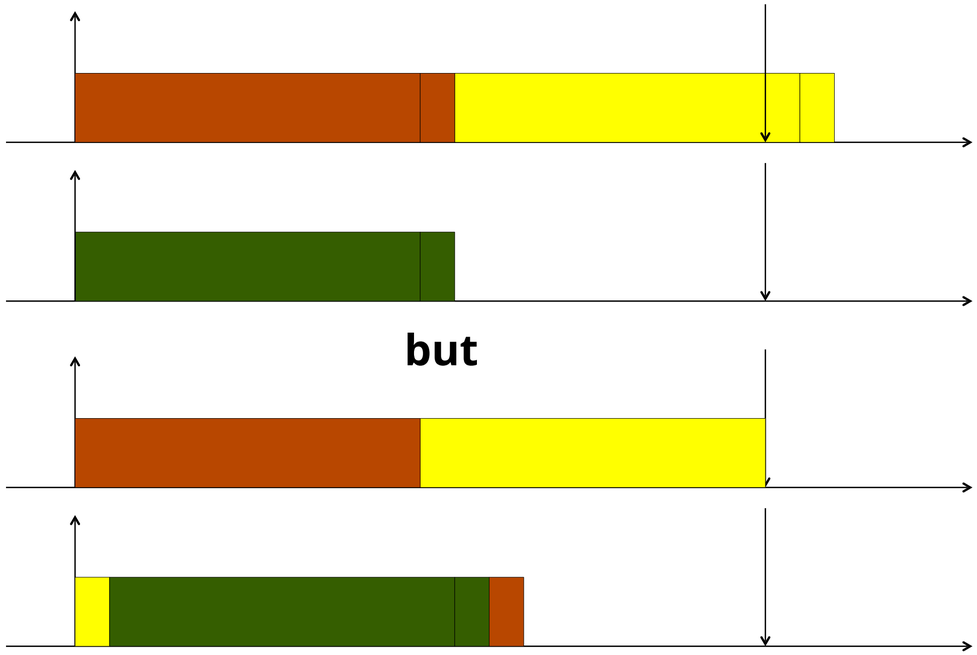
- always pick the ready jobs of the m most “important” tasks
- a task may migrate:
  when a new job of a task is released it may receive a different processor
- once started, a job is no longer migrated
- no need for task allocation to processors / load balancing



Global Scheduling: Utilization Bound

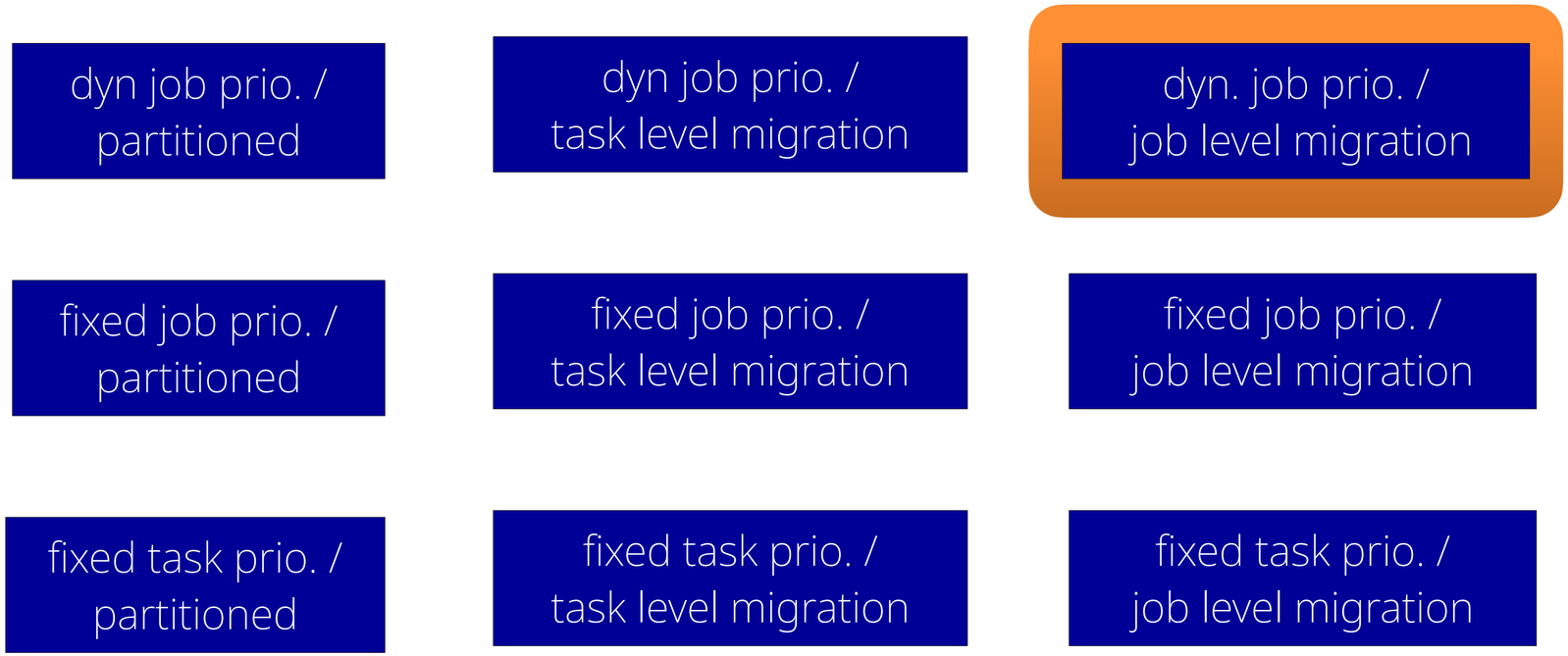
Utilization bound for global fixed-job priority algorithms

on m CPUs, G-FJP algorithms cannot schedule m+1 tasks with $C_i = 1 + \varepsilon$, $P_i = 2$



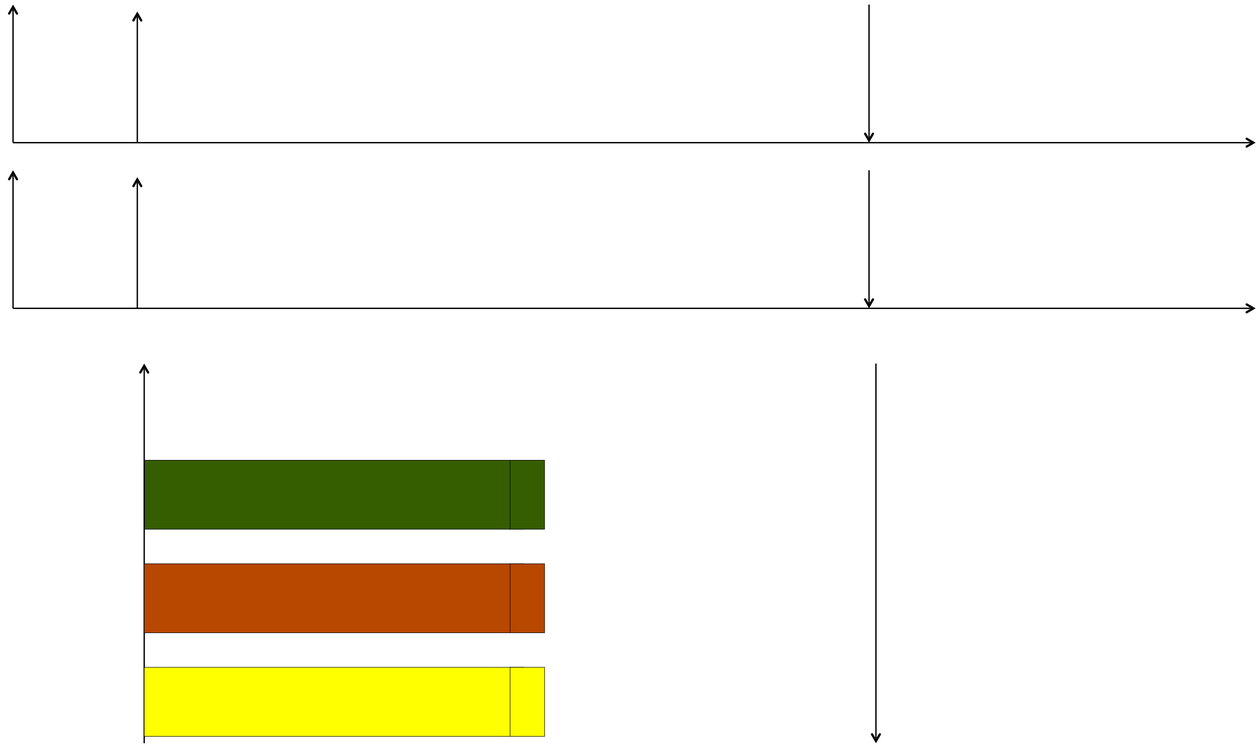
$$U_{opt} = \lim_{\varepsilon \to 0} \frac{(m+1)(1+\varepsilon)}{2} = \frac{m+1}{2}$$

Global Scheduling - Job Level Migration



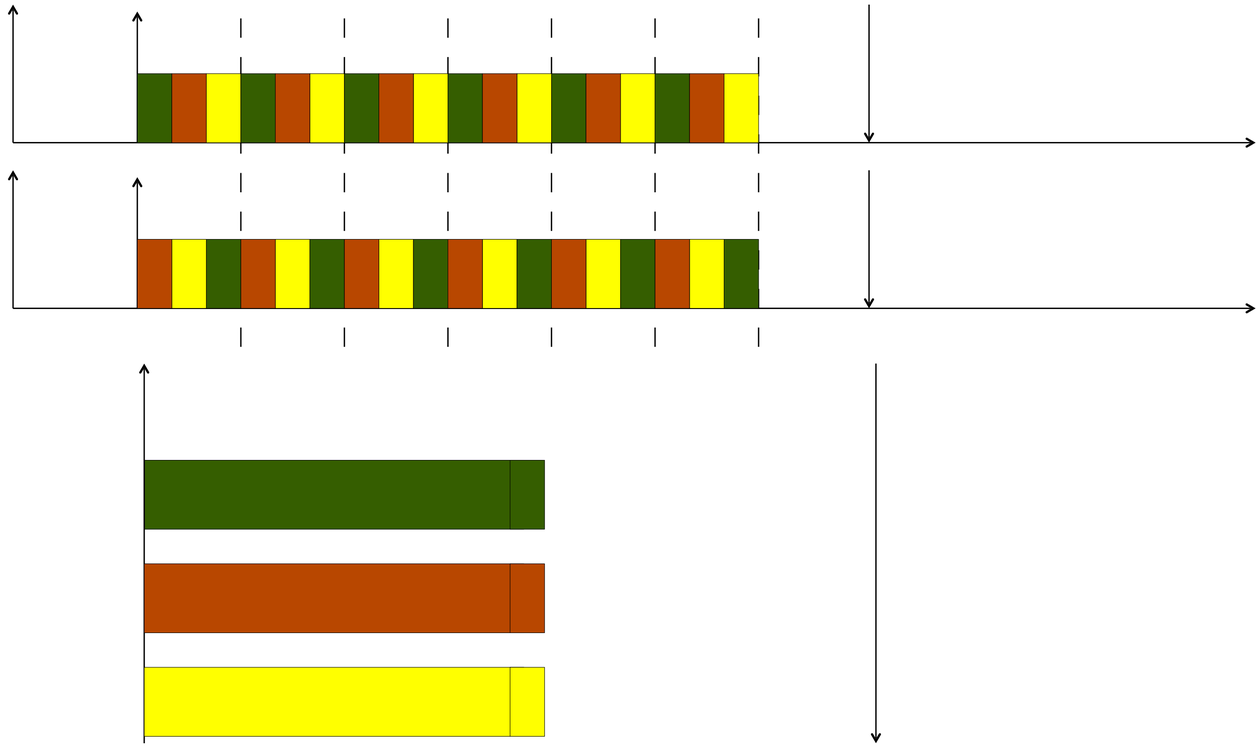
PFAIR — *Baruah '96*

- divide timeline into equal-length quanta
- at each quantum of length t, allocate tasks T_i to processors such that the accumulated processor time is $t \times u_i$



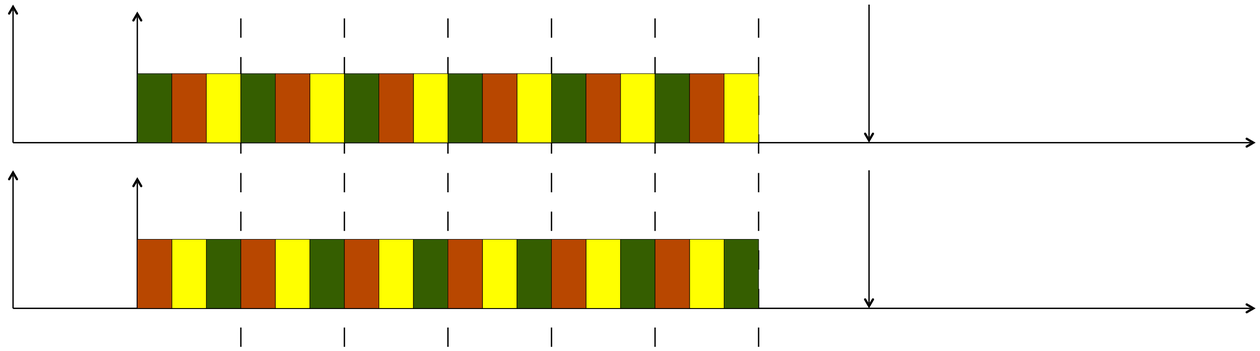
PFAIR — *Baruah '96*

- divide timeline into equal-length quanta
- at each quantum of length t, allocate tasks T_i to processors such that the accumulated processor time is $t \times u_i$



PFAIR — *Baruah '96*

- divide timeline into equal-length quanta
- at each quantum of length t, allocate tasks T_i to processors such that the accumulated processor time is $t \times u_i$



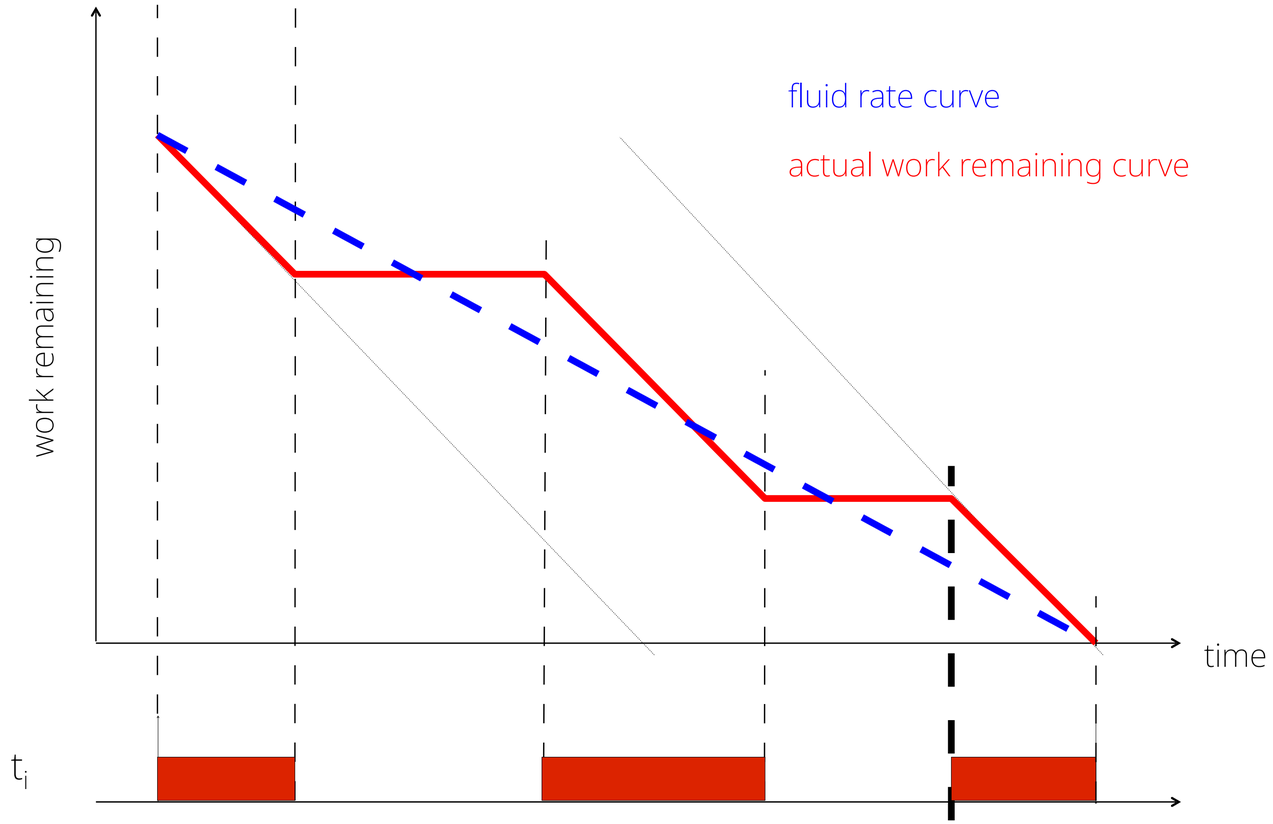
- PFAIR is optimal for periodic implicit deadline tasksets: $U_{OPT} = m$
- very high preemption and migration costs

DP-Fair — *Brandt '10*

Can we find an optimal scheduler for periodic implicit deadline task sets with a minimal number of preemptions / migrations?

- Recall *Hong '88*: there is no optimal multiprocessor scheduling algorithm for arbitrary task sets unless all tasks have the same relative deadline
- deadline partitioning: any task's deadline becomes a deadline for all tasks
- always run zero laxity jobs
  - laxity = time to deadline – remaining execution time
  - zero laxity → job may miss its deadline if it is not run now
- jobs that wind around the *fluid rate curve* are 'in good shape'

DP-Fair — *Brandt '10*



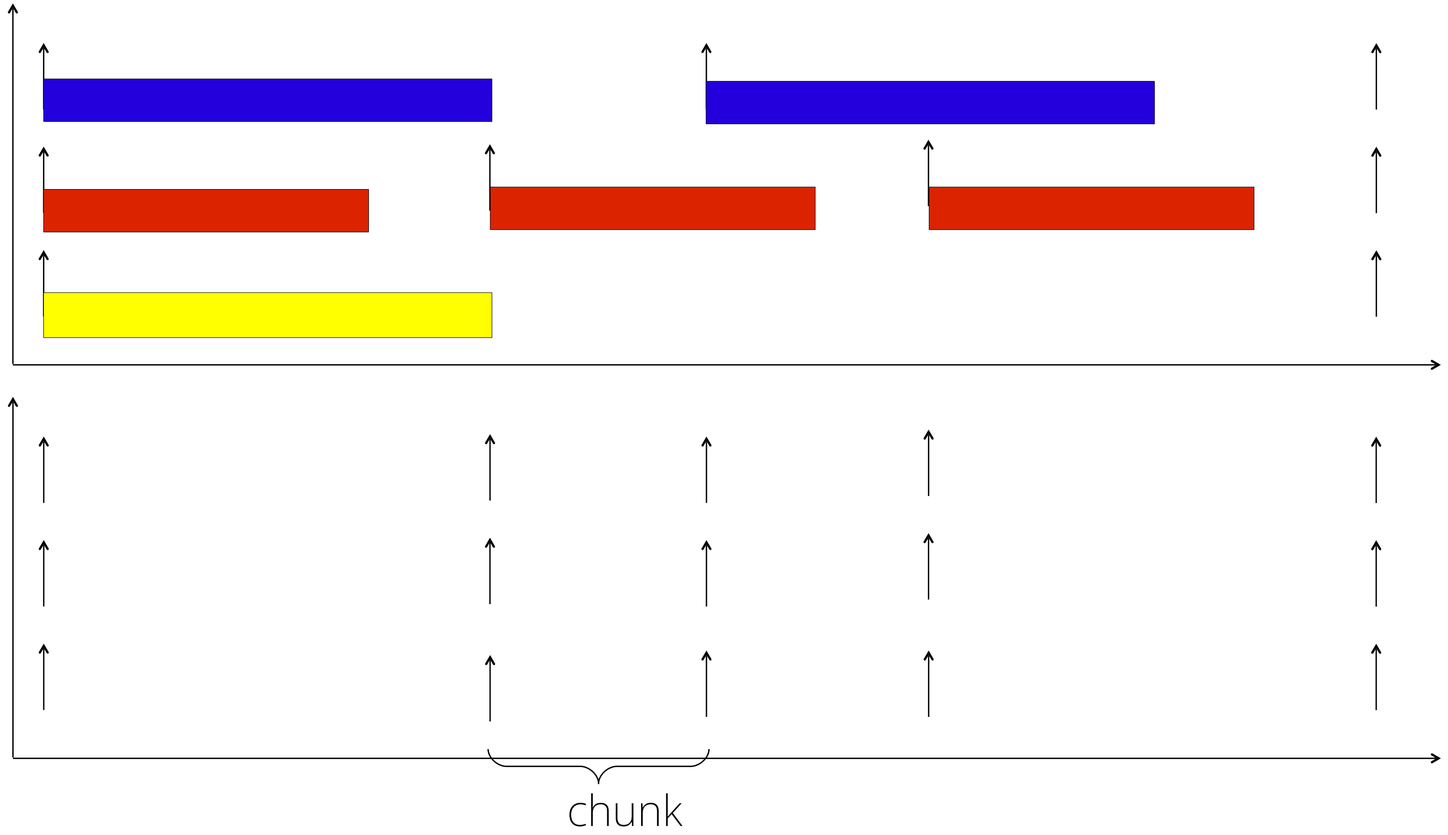
zero laxity event: no more time to run others

DP-Fair — *Brandt '10*

- a family of optimal, deadline partitioning scheduling algorithms
- split timeline into chunks according to job deadlines
- allocate work to a chunk proportional to U_i
  local execution time: $C^l_{i,j}=(t_{j+1} - t_j)\ U_i$
- Rule 1: always run a job with zero local laxity
- Rule 2: never run a job with no remaining local work
- Rule 3: split up idle time proportional to length of chunk

DP-Fair — *Brandt '10*

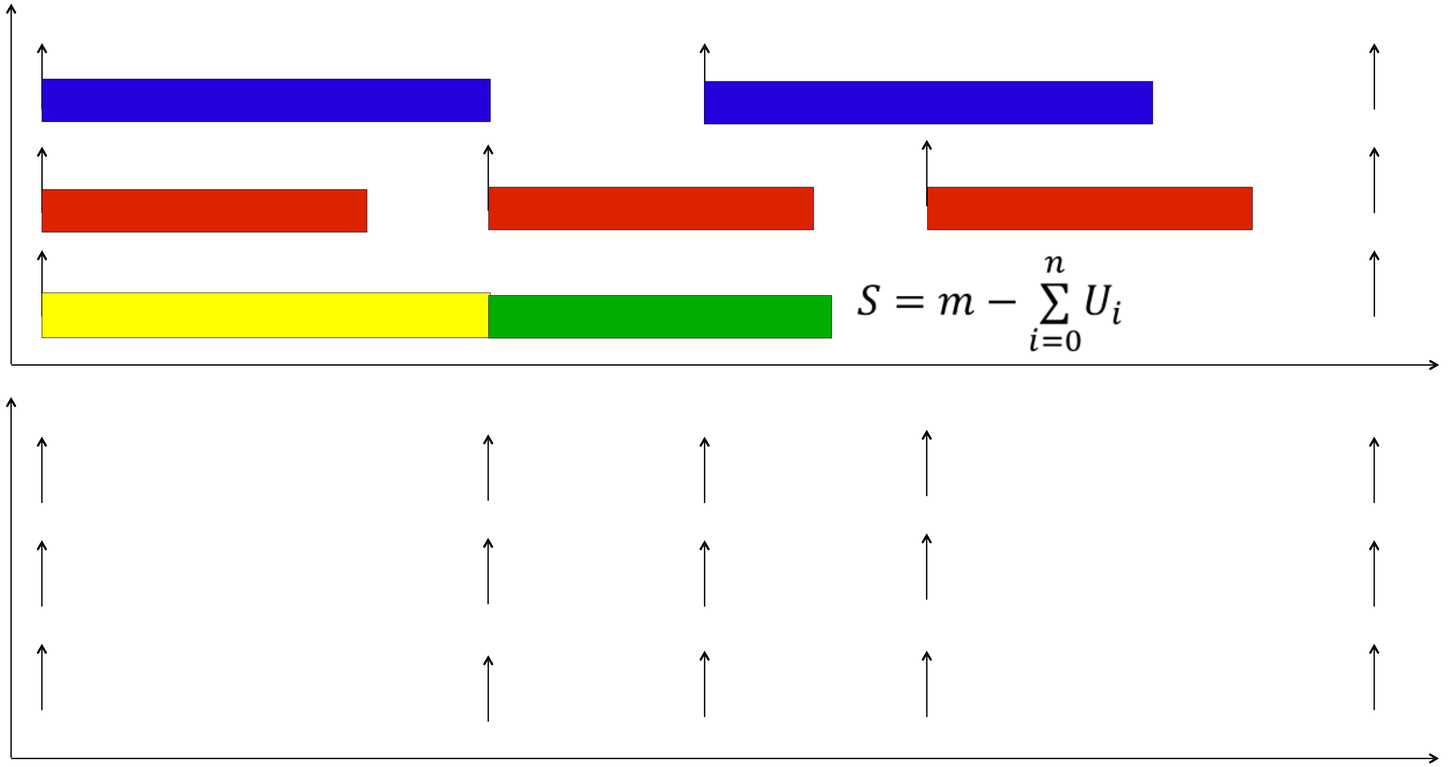
Deadline Partitioning



Introduce additional releases / deadlines for all jobs whenever there is such an event for one job in the original schedule => chunks

DP-Fair — *Brandt '10*

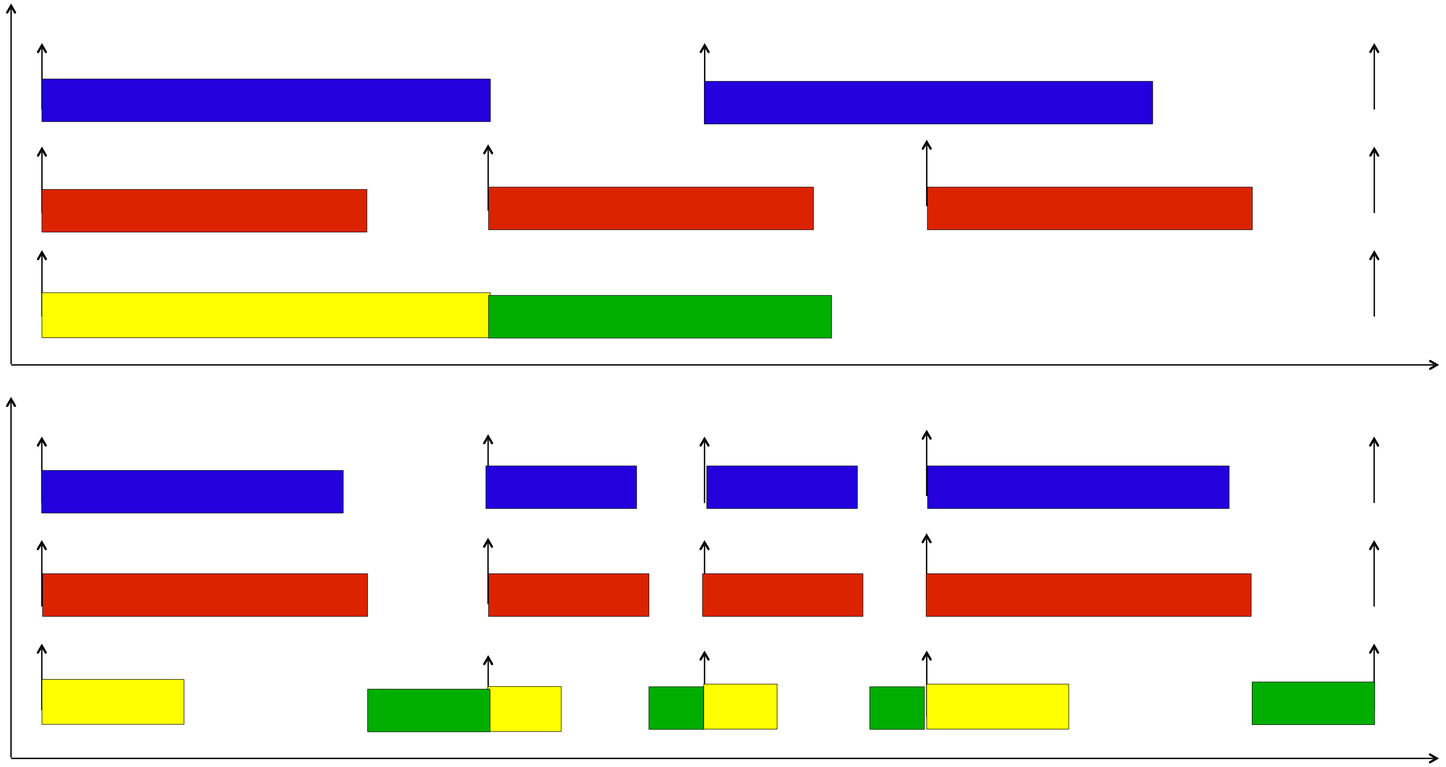
Idle time



Treat idle time as just another job to schedule.

DP-Fair — *Brandt '10*

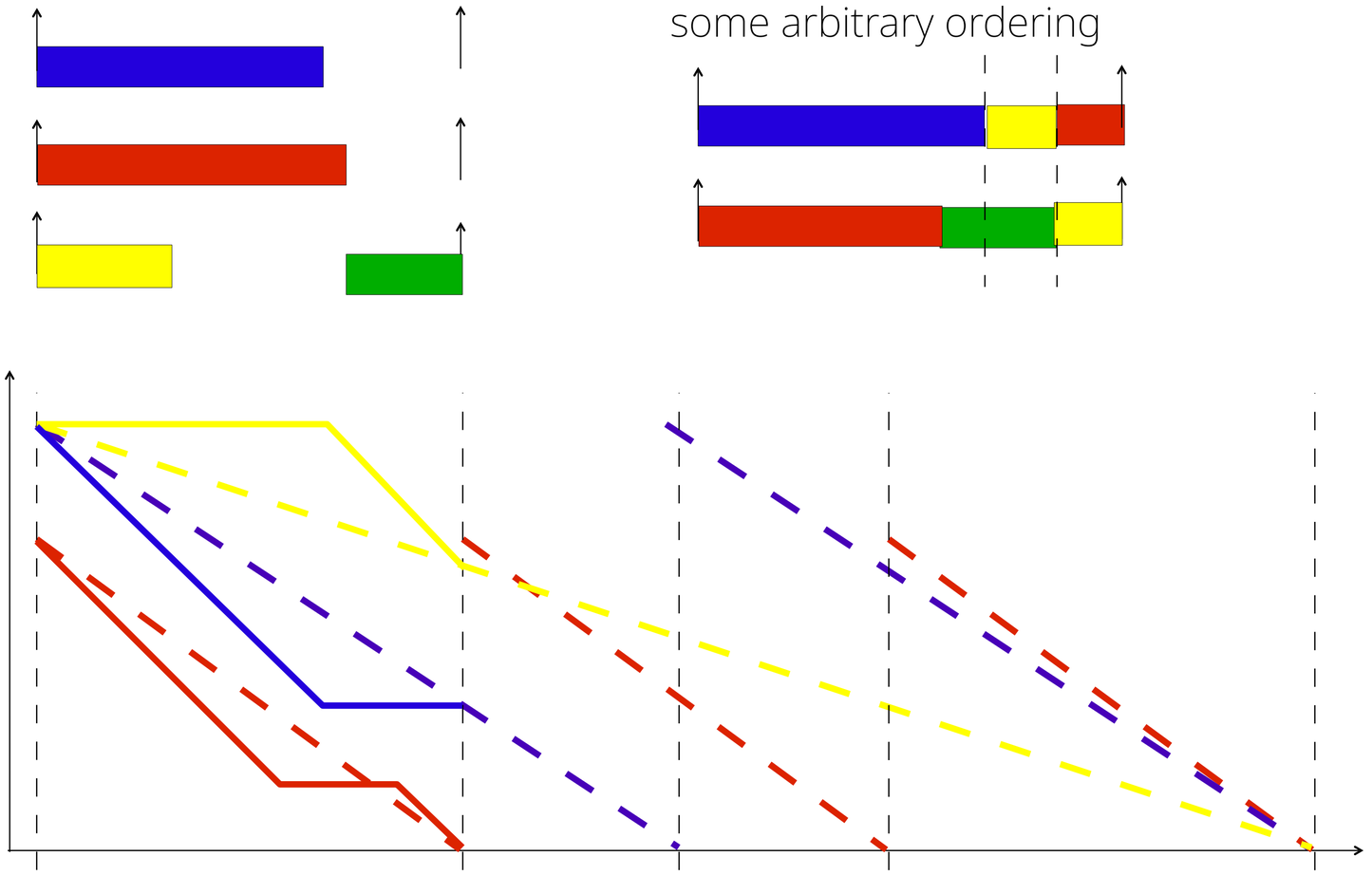
Allocate work proportional to U_i



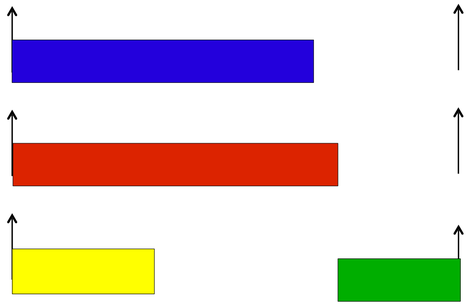
Allocate execution (and idle) time of a job proportionally to its utilization
=> amount of time that this job must run in a given chunk

DP-Fair — *Brandt '10*

Jobs hit their fluid rate curve at the end of each chunk

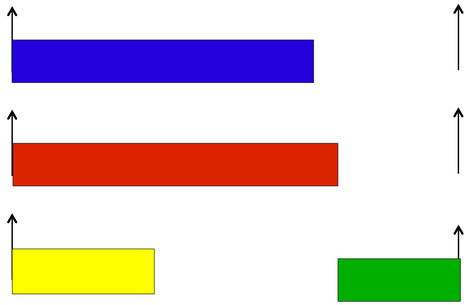


DP-Wrap – A DP-Fair Scheduler



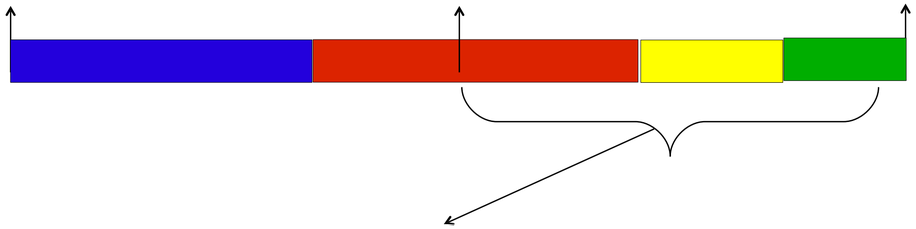
Work allocated to a chunk:
find a schedule

DP-Wrap – A DP-Fair Scheduler



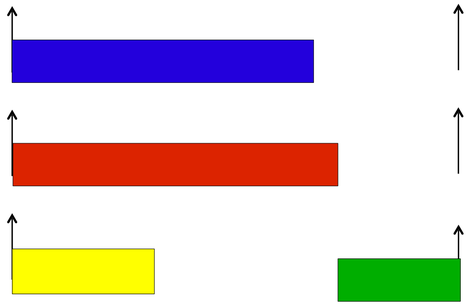
Work allocated to a chunk:
find a schedule

arrange jobs consecutively



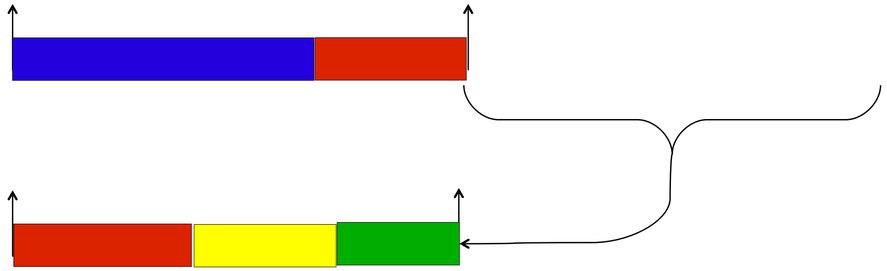
wrap around to obtain schedule for 2nd CPU, …

DP-Wrap – A DP-Fair Scheduler



Work allocated to a chunk:
find a schedule

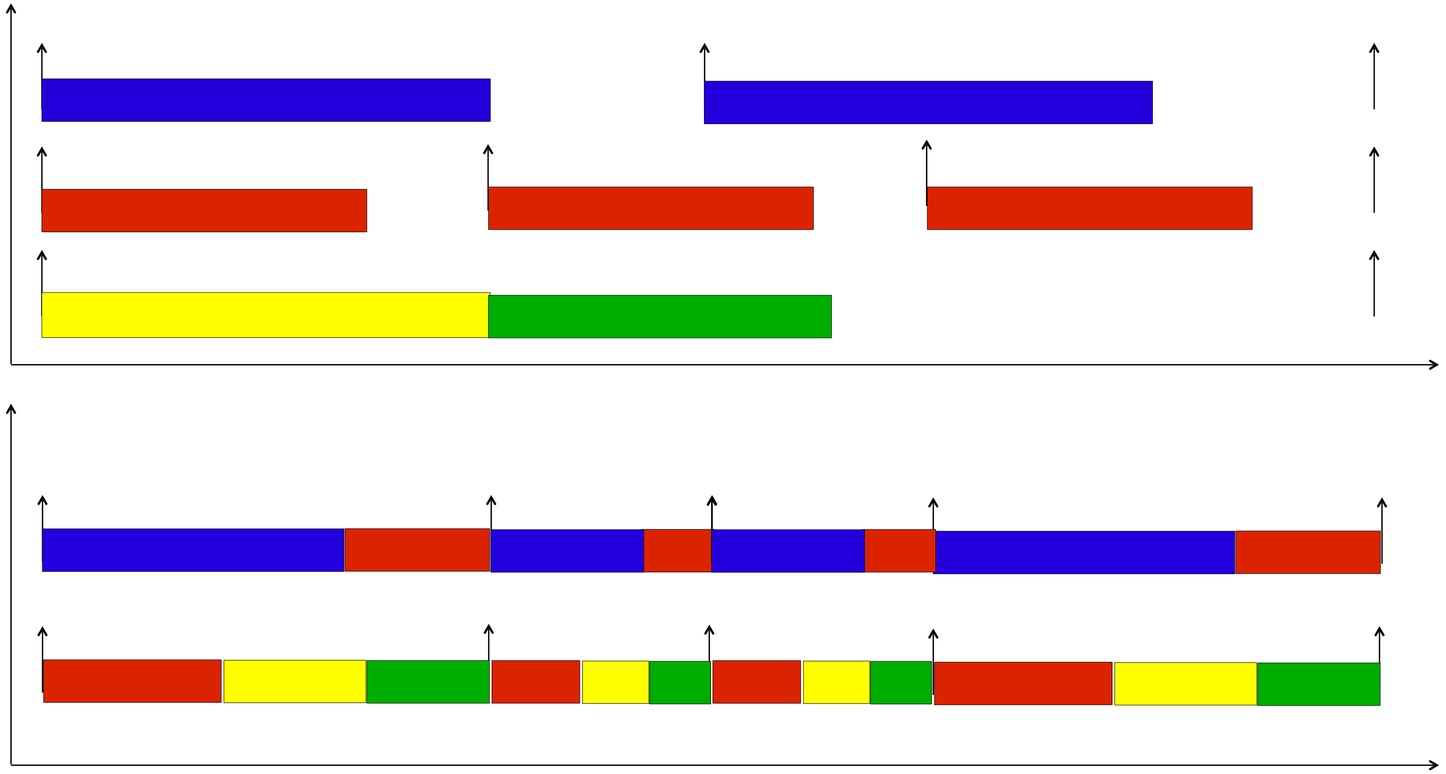
arrange jobs consecutively



wrap around to obtain schedule for 2nd CPU, …

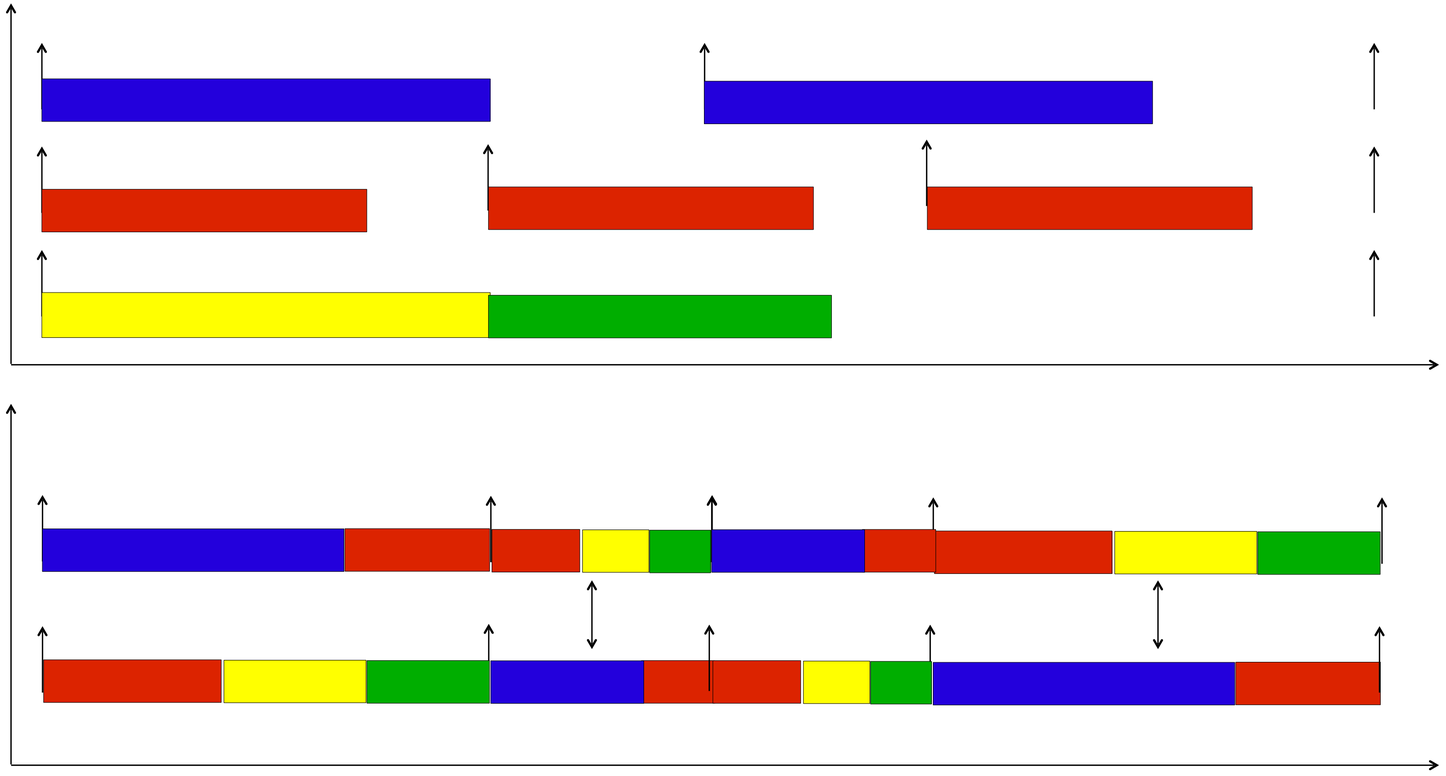
m – 1 migrations per chunk
n – 1 context switches per chunk

DP-Wrap – A DP-Fair Scheduler



Unnecessary migration of red task at chunk boundaries
=> mirror processor assignment of every second chunk

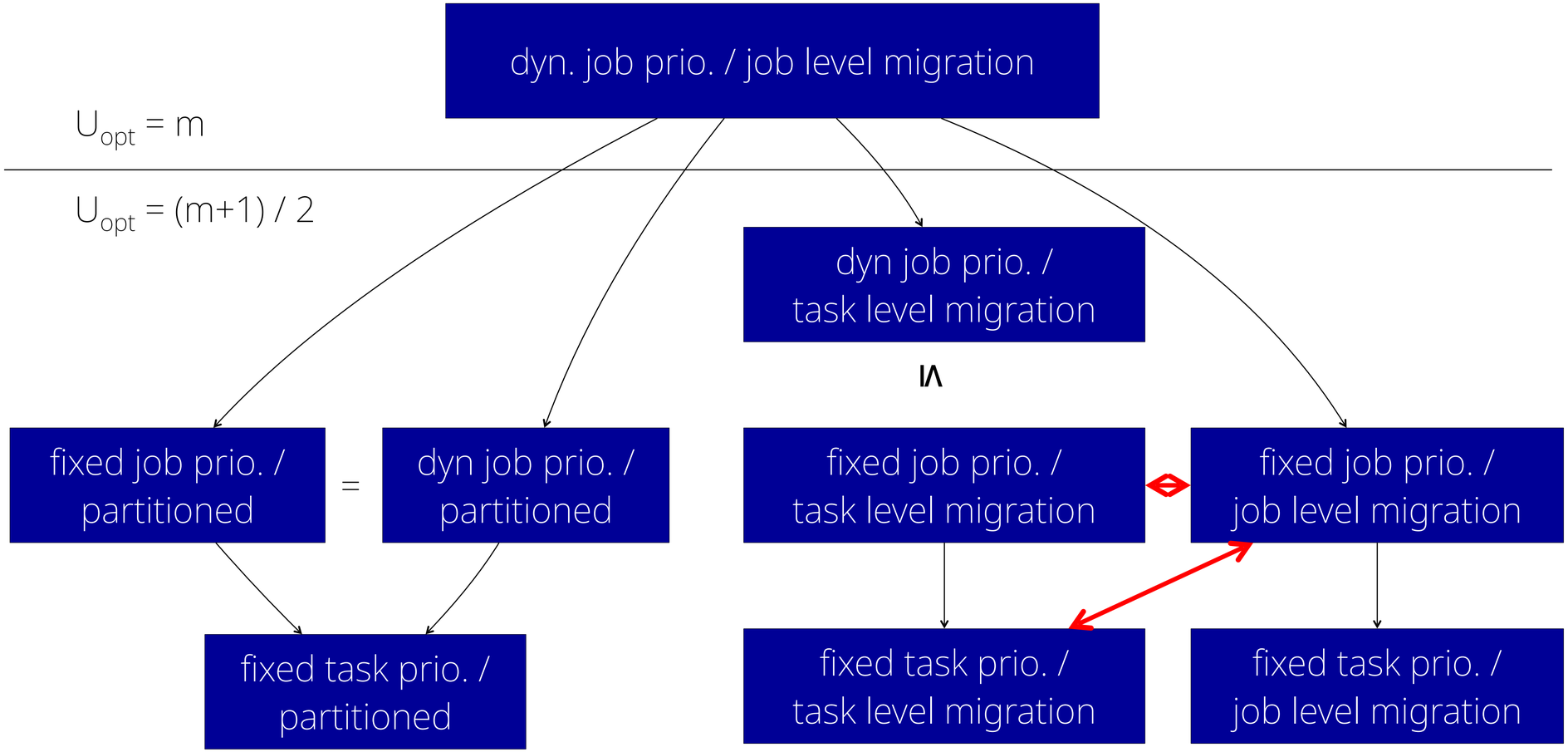
DP-Wrap – Mirrored



Design Space of Multiprocessor Scheduling

dyn job prio. / partitioned	dyn job prio. / task level migration	dyn. job prio. / job level migration
fixed job prio. / partitioned	fixed job prio. / task level migration	fixed job prio. / job level migration
fixed task prio. / partitioned	fixed task prio. / task level migration	fixed task prio. / job level migration

Design Space of Multiprocessor Scheduling



A → B => A can schedule any taskset that B can schedule and more

A ↔ B => dominance is not yet known