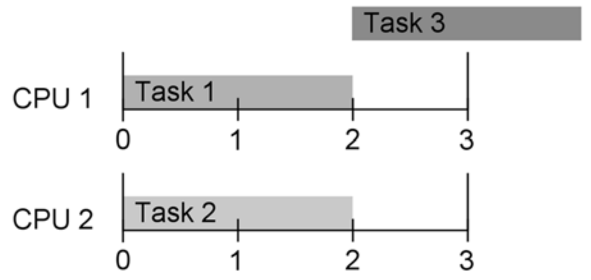
7.b Seeking the lost optimality

Partitioned Schedulers ≠ Optimal

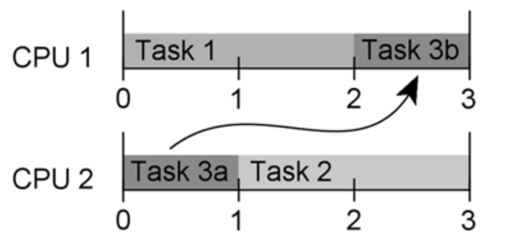
- Example: 2 processors; 3 tasks, each with 2 units of work required every 3 time units: $(3,2)$



Global Schedulers May Succeed

- Example: 2 processors; 3 tasks, each with 2 units of work required every 3 time units

Task 3 *migrates* between processors

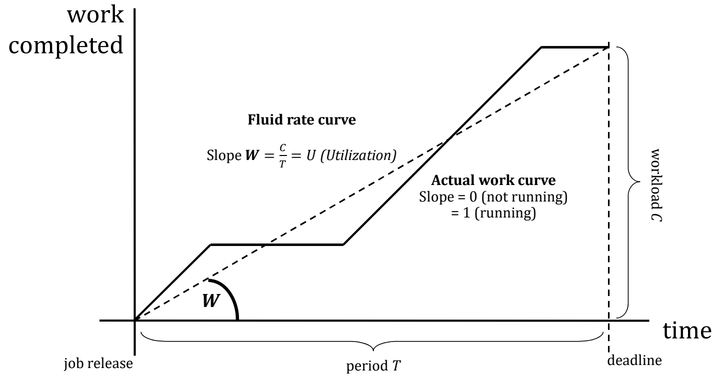


DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling

Greg Levin[†] Shelby Funk[‡] Caitlin Sadowski[†]
Ian Pye[†] Scott Brandt[†]

[†]University of California Santa Cruz
[‡]University of Georgia Athens

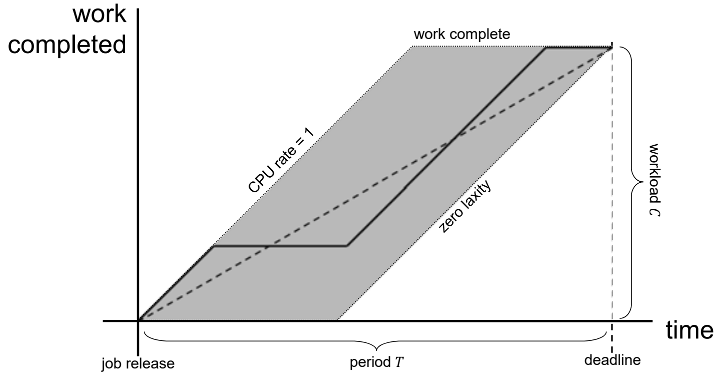
Fluid Rate Curve



The Grand Challenge (*Mark 1*)

- Design an *optimal* scheduling algorithm for periodic task sets on *multiprocessors*
  - ☐ A task set is *feasible* if there exists a schedule that meets all deadlines
  - ☐ A scheduler is *optimal* if it can always schedule any feasible task set

Feasible Work Region



Necessary and Sufficient Conditions

- Any set of (independent) tasks needing at most
  - ☐ 1 processor for each task τ_i ($\forall i\ U_i \leq 1$)
  - ☐ m processors for all tasks ($\sum_i U_i \leq m$)

  is feasible
- **Proof**: small scheduling intervals can approximate the fluid rate curve (at what cost?)
  - ☐ **Status**: solved. P-Fair (1996) was the first optimal algorithm

The Grand Challenge (*Mark 2*)

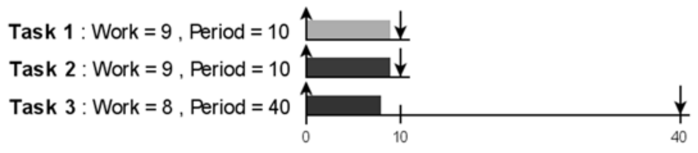
- Design an *optimal* scheduling algorithm with *fewer* context switches and migrations
  - Finding a feasible schedule with *the fewest* migrations is NP-Complete!

The Grand Challenge (*Mark 2*)

- Design an *optimal* scheduling algorithm with *fewer* context switches and migrations
- Status: *Solved*
  - **BUT** the solutions are complex and confusing
- **Our Contributions:** A *simple, unifying theory* for optimal global multiprocessor scheduling *and* a simple optimal algorithm

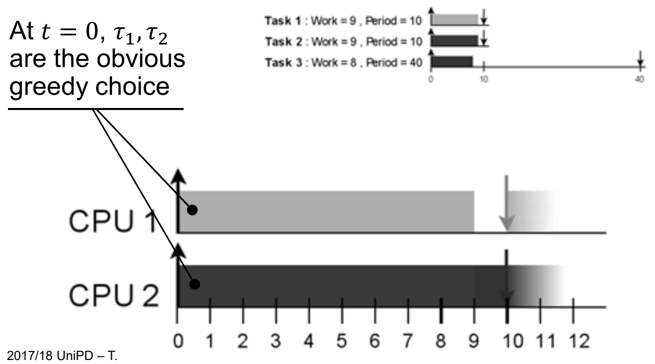
Why Greedy Algorithms Fail On Multiprocessors

- Example ($n = 3$, $m = 2$)


Task 1 : Work = 9 , Period = 10
Task 2 : Work = 9 , Period = 10
Task 3 : Work = 8 , Period = 40
0
10
40

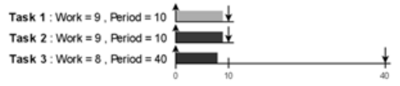
Utilization: 9/10 + 9/10 + 8/40 = 2

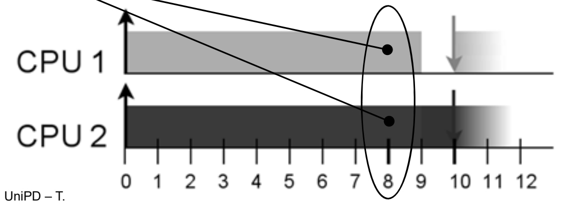
Why Greedy Algorithms Fail On Multiprocessors


At $t = 0$, τ_1, τ_2 are the obvious greedy choice
Task 1 : Work = 9 , Period = 10
Task 2 : Work = 9 , Period = 10
Task 3 : Work = 8 , Period = 40
CPU 1
CPU 2
0 1 2 3 4 5 6 7 8 9 10 11 12

Why Greedy Algorithms Fail On Multiprocessors

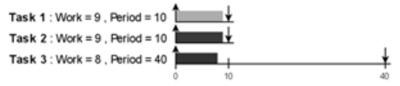
Even at $t = 8$, τ_1, τ_2 are the only "reasonable" greedy choice

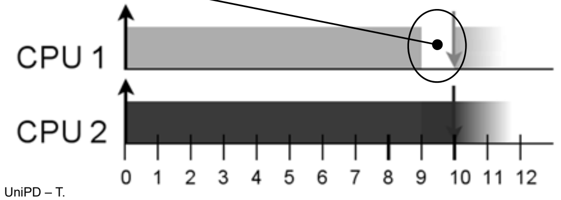




Why Greedy Algorithms Fail On Multiprocessors

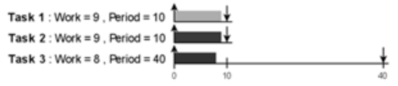
Yet, if τ_3 isn't started by $t = 8$, the resultant idle time eventually causes a deadline miss

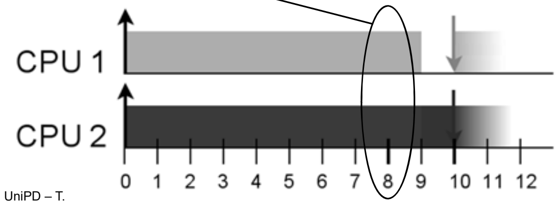




Why Greedy Algorithms Fail On Multiprocessors

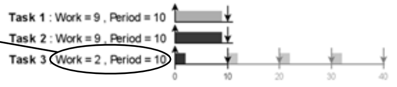
How can we "see" this critical event at $t = 8$?

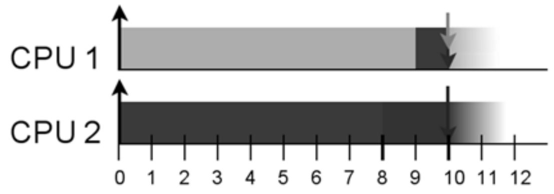




Proportioned Algorithms Succeed On Multiprocessors

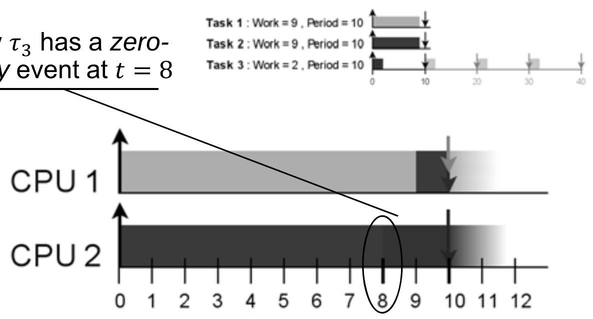
Subdivide τ_3 in two subtasks with the same period as τ_1, τ_2





Proportioned Algorithms Succeed On Multiprocessors

Now τ_3 has a *zero-laxity* event at $t = 8$



Proportional Fairness

- **Insight:** scheduling is easier when all jobs have the same deadline

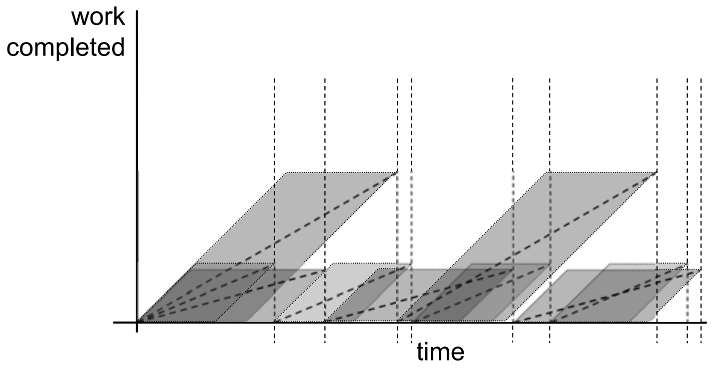
Theorem [Hong, Leung: RTSS 1988, IEEE TCO 1992]
No optimal on-line scheduler can exist for a set of jobs with two or more distinct deadlines on any m multiprocessor system, where m > 1

- **Application:** apply all deadlines to all jobs
  - Assign workloads proportional to utilization
  - Work complete matches fluid rate curve at every system deadline

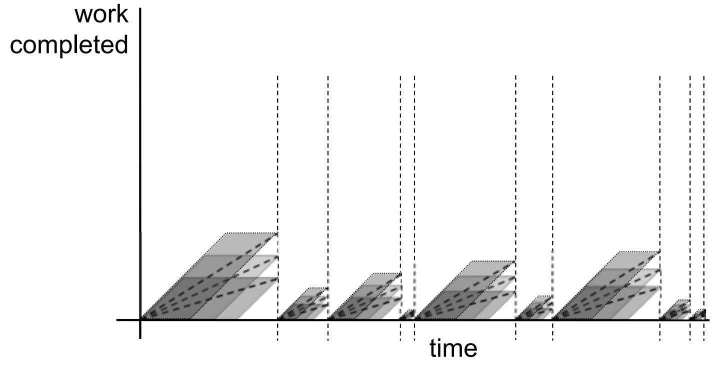
Proportional Fairness is the Key

- All known optimal algorithms enforce proportional fairness at all deadlines
  - **P-Fair** (1996) - *Baruah, Cohen, Plaxton, and Varvel*
    (the extreme: proportional fairness at *all times*)
  - **BF** (2003) - *Zhu, Mossé, and Melhem*
  - **LLREF** (2006) - *Cho, Ravindran, Jensen*
  - **EKG** (2006) - *Andersson, Tovar*
- Why do they all use proportional fairness?

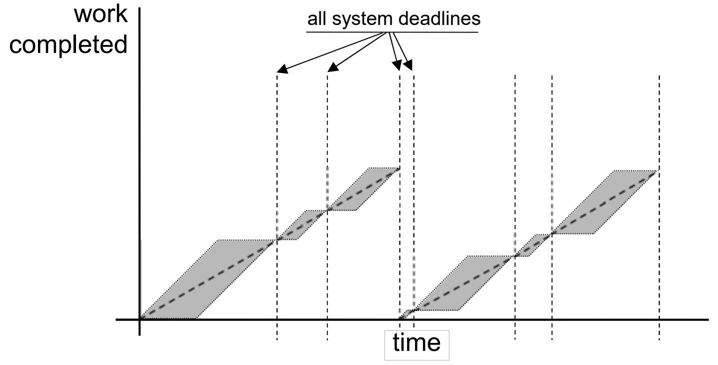
Scheduling Multiple Tasks is Complicated



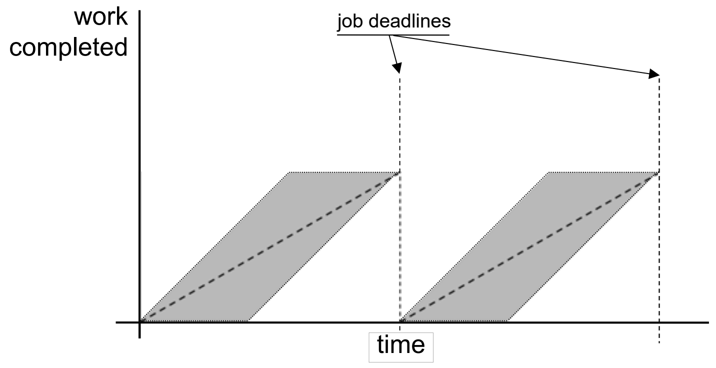
Scheduling Multiple Tasks with Same Deadline is Easy



Restricted Feasible Regions Under Deadline Partitioning



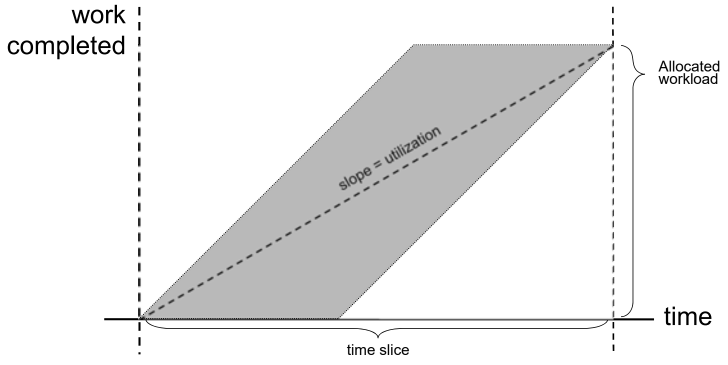
Actual Feasible Regions



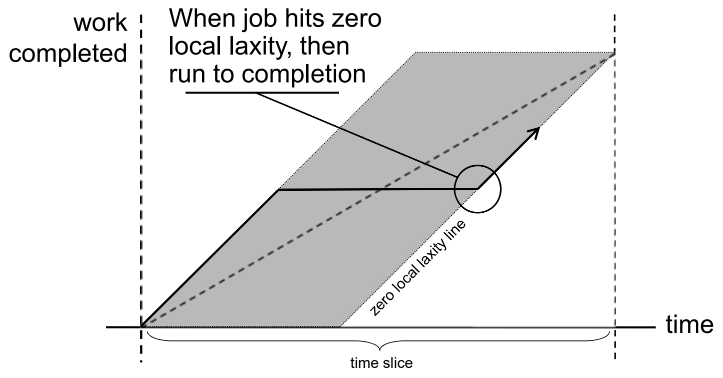
The DP-Fair Scheduling Policy

- Partition time into *slices* based on all system deadlines
- Allocate each job a per-slice workload equal to its *utilization* times the *length of the slice*
- Schedule jobs within each slice in any way that obeys the following three rules:
  1. Always run a job with zero *local laxity*
  2. Never run a job with no workload remaining in the slice
  3. Do not voluntarily allow more idle processor time than $(m - \sum U_i) \times$ (*length of slice*)

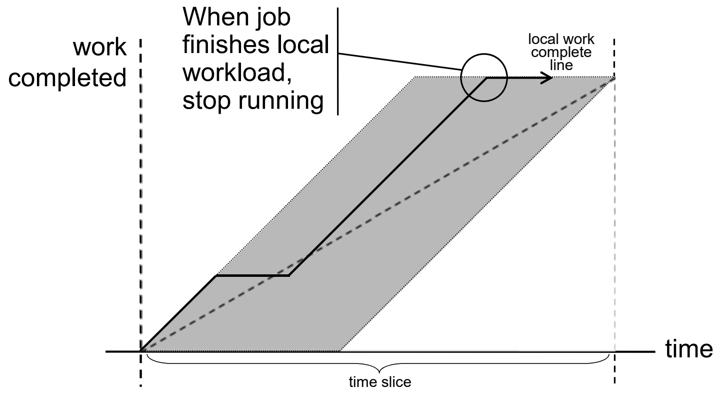
DP-Fair Work Allocation



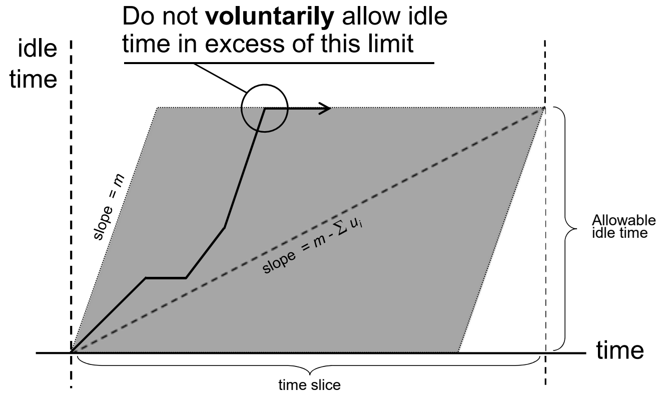
DP-Fair Scheduling Rule #1



DP-Fair Scheduling Rule #2



DP-Fair Scheduling Rule #3



DF-Fair Guarantees Optimality

- We say that a scheduling algorithm is *DP-Fair* if it follows these three rules
- **Theorem:** Any DP-Fair scheduling algorithm for periodic tasks is optimal

EXAMPLE OF EXAM ASSIGNMENT: STUDYING THE RUN ALGORITHM

PhD seminar on Real-Time Systems, University of Bologna, July 2014

DP-Fair Implications

- (Partition time into slices)
  + (Assign proportional workloads)

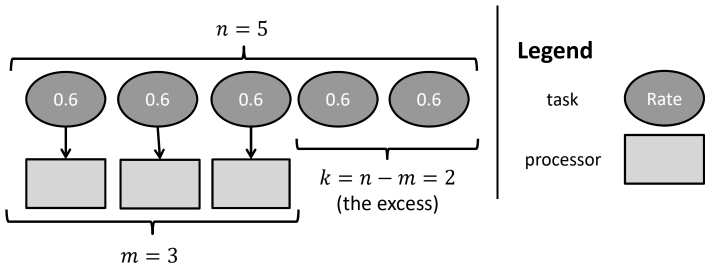
  Optimal scheduling is almost trivial
  - Minimally restrictive rules allow great latitude for algorithm design and adaptability
- What is the simplest possible algorithm?

RUN Assumptions

Model parameters

- m homogeneous (symmetric) processors
- Implicit-deadline independent task $\tau_i, i \in \{1..n\}$
- $n = m + k, k \geq 0$
- Fixed-rate tasks $U_i = \frac{C_i}{T_i}$ $\sum_{i=1}^{n} U_i \leq m$
- Fully utilized system: no idle time (perhaps using fillers)
- *Migration* and *preemption* are assumed to have no additional costs over c_i

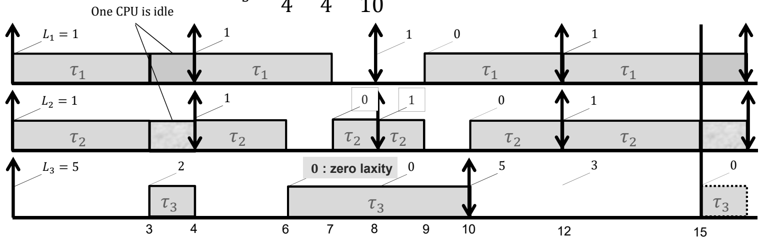
Example /1



- $U_i = 0.6 \;\; \forall \tau_i, i = \{1, \dots, n = 5\}$
- $\sum_{i=1}^{n=5} U_i = 3 \Rightarrow m = 3$ (fully utilized system)
- What schedule Σ for $\mathbf{S} = \{\{\boldsymbol{\tau_i}\}, \boldsymbol{m}\}$?

The G-LLF example at page 429 …

$$S = \{\tau_1 = (3{,}4), \tau_2 = (3{,}4), \tau_3 = (5{,}10)\}, H_S = 20$$
$$U_S = \frac{3}{4} + \frac{3}{4} + \frac{5}{10} = 2.0 \;\rightarrow m = 2$$



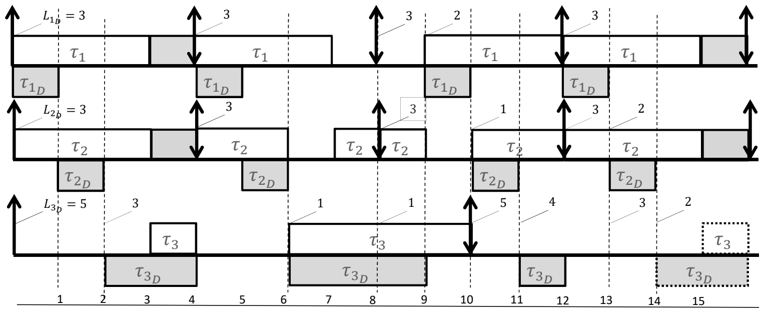
- At $t = 15$ the CPU time remaining is $T_R = m \times (H_S - t) = \mathbf{10}$
- Yet, the time needed is $T_N = e_1 + e_2 + e_3 = \mathbf{11}$

Duality

- The problem of scheduling
  $\mathbf{S} = \{\boldsymbol{\tau_1} = (\boldsymbol{c_1}, \boldsymbol{T_1}), \dots, \boldsymbol{\tau_n} = (\boldsymbol{c_n}, \boldsymbol{T_n})\}, \boldsymbol{m}$
  has a *dual* problem that consists of scheduling
  $\boldsymbol{S'} = \{\boldsymbol{\tau'_1} = (\boldsymbol{T_1} - \boldsymbol{c_1}, \boldsymbol{T_1}), \dots, \boldsymbol{\tau'_n} = (\boldsymbol{T_n} - \boldsymbol{c_n}, \boldsymbol{T_n})\}, (\boldsymbol{n} - \boldsymbol{m})$
- With this definition of duality
  - Laxity in primal is work remaining in the dual
  - A work-complete event in the primal is zero-laxity in the dual
  - And viceversa
- **Corollary**: any scheduling problem with $\boldsymbol{m}$ processors and $\boldsymbol{n} = \boldsymbol{m} + \mathbf{1}$ tasks and $\sum_1^n U_i = \boldsymbol{m}$ may be scheduled by applying EDF to its uniprocessor dual
  - If I can schedule n tasks on m processors, then I can also schedule the same n tasks on $n - m$ processors
  - This is so because the scheduling events in the dual map to scheduling events in the primal

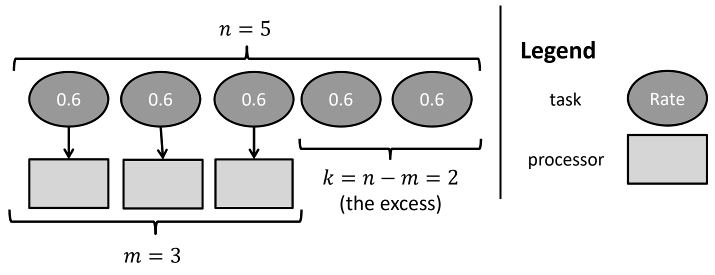
Applying duality

$$S = \{\tau_1 = (3{,}4), \tau_2 = (3{,}4), \tau_3 = (5{,}10)\}, U_s = \frac{3}{4} + \frac{3}{4} + \frac{5}{10} = 2.0 \;\rightarrow m = 2$$
$$\boldsymbol{S_D} = \{\boldsymbol{\tau_{1_D}} = (\mathbf{1}, \mathbf{4}), \boldsymbol{\tau_{2_D}} = (\mathbf{1}, \mathbf{4}), \boldsymbol{\tau_{3_D}} = (\mathbf{5}, \mathbf{10})\}, \boldsymbol{U_{S_D}} = \frac{\mathbf{1}}{\mathbf{4}} + \frac{\mathbf{1}}{\mathbf{4}} + \frac{\mathbf{5}}{\mathbf{10}} = \mathbf{1.0} \rightarrow \boldsymbol{m_D} = \mathbf{1}$$



The dual (LLF) schedule leaves no idle time

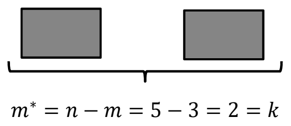
Example /1



- $U_i = 0.6 \;\; \forall \tau_i, i = \{1, \dots, n = 5\}$
- $\sum_i^n U_i = 3 \Rightarrow m = 3$ (fully utilized system)
- What schedule Σ for $\mathbf{S} = \{\{\boldsymbol{\tau_i}\}, \boldsymbol{m}\}$?

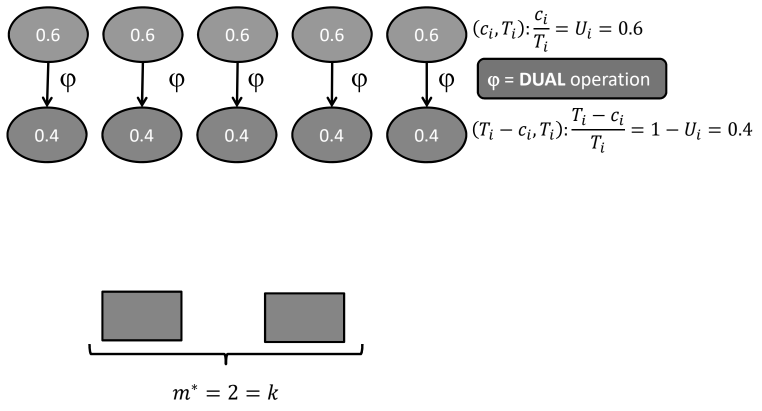
Example /2

- Consider the dual of this $\{n = 5, m = 3\}$ system

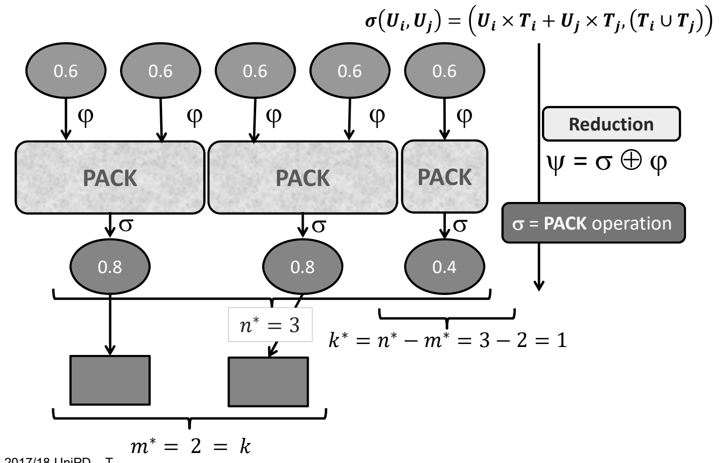


The dual should run on $m^* = 2$ processors

Example /3

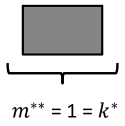

0.6
0.6
0.6
0.6
0.6
φ
φ
φ
φ
φ
0.4
0.4
0.4
0.4
0.4
$(c_i, T_i): \frac{c_i}{T_i} = U_i = 0.6$
φ = DUAL operation
$(T_i - c_i, T_i): \frac{T_i - c_i}{T_i} = 1 - U_i = 0.4$
$m^* = 2 = k$

Example /4

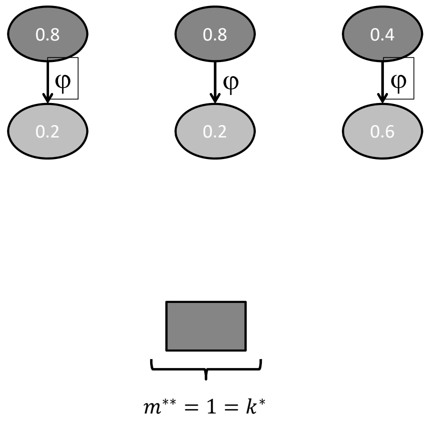

$\boldsymbol{\sigma}(\boldsymbol{U_i}, \boldsymbol{U_j}) = (\boldsymbol{U_i} \times \boldsymbol{T_i} + \boldsymbol{U_j} \times \boldsymbol{T_j}, (\boldsymbol{T_i} \cup \boldsymbol{T_j}))$
0.6
0.6
0.6
0.6
0.6
φ
φ
φ
φ
φ
PACK
PACK
PACK
σ
σ
σ
0.8
0.8
0.4
Reduction
ψ = σ ⊕ φ
σ = PACK operation
$n^* = 3$
$k^* = n^* - m^* = 3 - 2 = 1$
$m^* = 2 = k$

Example /5

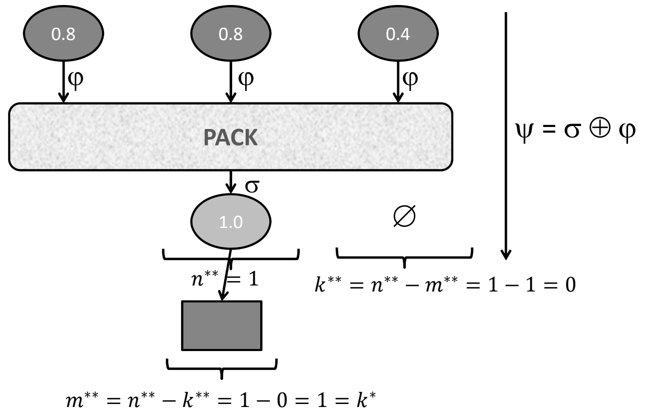
The $(n^* = 3, m^* = 2)$ system still cannot be partitioned feasibly
Yet, applying duality to it seems promising since the dual would need $n^* - m^* = 1$ processor, which would REDUCE the problem TO a UNIPROCESSOR case



Example /6

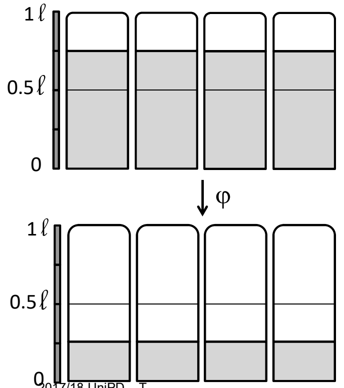


Example /7


0.8
0.8
0.4
φ
φ
φ
PACK
ψ = σ ⊕ φ
σ
1.0
∅
$n^{**} = 1$
$k^{**} = n^{**} - m^{**} = 1 - 1 = 0$
$m^{**} = n^{**} - k^{**} = 1 - 0 = 1 = k^*$

Why does reduction terminate? /1

Lemma: $\psi = \left|\sigma \circ \varphi\left(\bigcup_1^4 \tau_i\right)\right| \leq \left\lceil \frac{|\tau|+1}{2} \right\rceil$



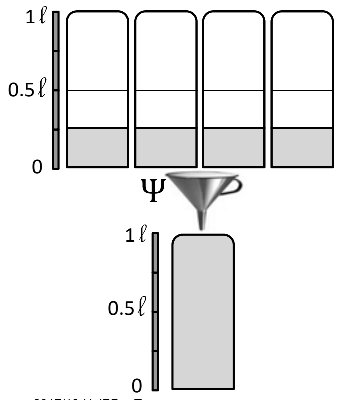
Intuition

$\sum_1^4 U_i = 3 \Rightarrow m = 3$
$n = 4$
$\boldsymbol{k} = n - m = 1$

In the dual system
$\sum_1^4 U_i^* = n - m = 1 \Rightarrow$
$m^* = 1 = \boldsymbol{k}$
$n^* = 1$ after packing
$k^* = 0$ no leftover

Why does reduction terminate? /2

Lemma: $\psi = \left|\sigma \circ \varphi\left(\bigcup_1^4 \tau_i\right)\right| \le \left\lceil \frac{|\tau|+1}{2} \right\rceil$

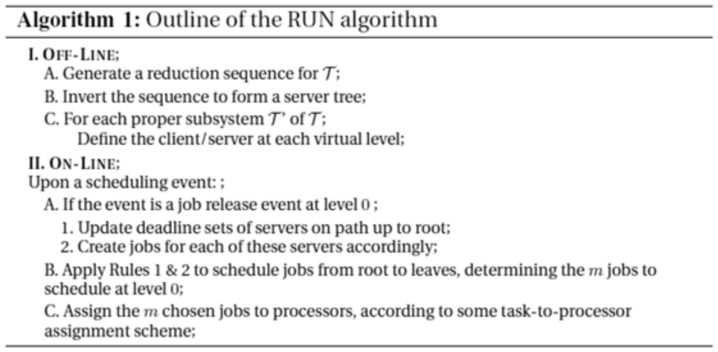


- Reduction $\psi = (\sigma \oplus \varphi)$ terminates as every step of it lowers the residual workload and the # of processors needed to run it
- The packing operation (at least) halves the number of tasks to schedule
- **Termination theorem:** after a finite number ***p*** of reduction steps, the system is reduced to a uniprocessor with full workload

How does RUN work /1

- A pair of basic operators
  - DUAL (φ)
  - PACK (σ)
- The REDUCE ($\psi = \sigma \oplus \varphi$) operation lowers (~ halves) the size of the problem at every step
- **Theorem** (validity of the dual): Σ valid $\Leftrightarrow \Sigma^*$ valid
- Since every dual task represents the idle time of its primary, finding a feasible schedule for the dual (which is easier) determines a feasible schedule for its primary

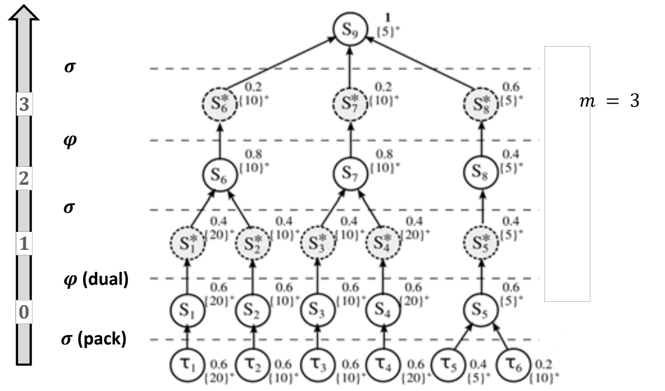
How does RUN work /2



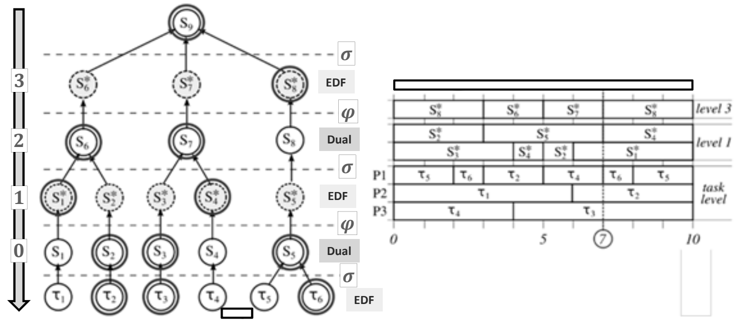
```
Algorithm 1: Outline of the RUN algorithm
I. OFF-LINE;
  A. Generate a reduction sequence for $\mathcal{T}$;
  B. Invert the sequence to form a server tree;
  C. For each proper subsystem $\mathcal{T}'$ of $\mathcal{T}$;
       Define the client/server at each virtual level;
II. ON-LINE;
Upon a scheduling event: ;
  A. If the event is a job release event at level 0 ;
    1. Update deadline sets of servers on path up to root;
    2. Create jobs for each of these servers accordingly;
  B. Apply Rules 1 & 2 to schedule jobs from root to leaves, determining the $m$ jobs to
  schedule at level 0;
  C. Assign the $m$ chosen jobs to processors, according to some task-to-processor
  assignment scheme;
```

Example: off-line phase

$S_5 = \sigma(U_4, U_5) = \left((0.4 + 0.2) \times \min(5,10)\right), (5 \cup 10)\right) = (0.6, 5), C_{S_5} = 3$



Example: on-line phase (at time $t = 7$)



RUN implementation

- **For real**
  - On top of **LITMUS^{RT}** Linux test-bed (UNC, now MP-SWI)
  - Relying on *standard* RTOS support
- Main implementation choices and challenges
  - *Scheduling on the reduction tree*
    - How to organize the data structure
    - How to perform virtual scheduling and trigger tree updates
    - Intrinsic influence of the packing policy
  - *Mixing global and local scheduling*
    - Global release event queue vs. local *level-0* ready queue
    - Handling simultaneous scheduling events
      - Job release, budget exhaustion (possibly from different sub-trees)
  - *Meeting the full-utilization requirement*
    - Variability of tasks' WCET and less-than-full utilization

PROXIMA

PROXIMA

Putting RUN into practice

Implementation and evaluation

Davide Compagnin, Enrico Mezzetti and Tullio Vardanega
University of Padua, Italy

26th EUROMICRO Conference on Real-time Systems (ECRTS)
Madrid, 9 July 2014

This project and the research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7 / 2007-2013] under grant agreement 611085

www.proxima-project.eu

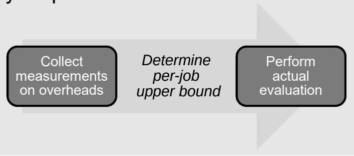
Empirical evaluation

- **Empirical evaluation** instead of simulation-based
- Focus on **scheduling interference**
  - Cost of scheduling primitives
  - Incurred preemptions and migrations
- RUN compared against **P-EDF** and **G-EDF**
  - RUN shares something in common with both
  - Much better than **Pfair** (S-PD^2 in LITMUS^{RT})
    - RUN has superior performance for preemptions and migrations

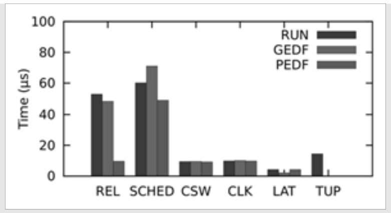
PROXIMA

Experimental setup

- ❑ **LITMUS^{RT}** on an 8-core AMD Opteron^{TM} 2356
- ❑ Collected measurements for RUN, P-EDF, G-EDF
  - ➢ Hundreds of automatically generated task sets
  - ➢ Harmonic and non-harmonic, with global utilization @ 50%-100%
  - ➢ Representative of small up to large tasks
- ❑ **Two-step process**
  - ➢ Preliminary empirical determination of overheads


Collect measurements on overheads
Determine per-job upper bound
Perform actual evaluation

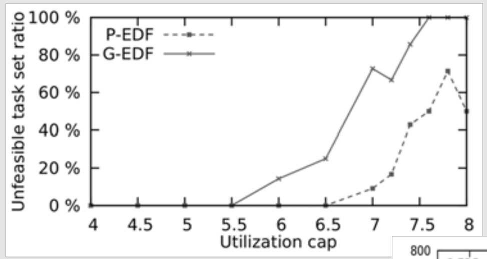
Primitive overheads and empirical bound

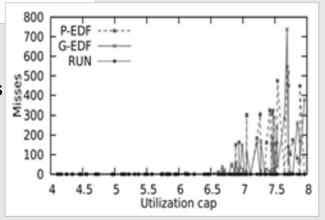


- ❑ Expectations confirmed
  - ➢ P-EDF needs lighter-weight scheduling primitives
- ❑ **Tree update** (TUP) triggered upon
  - ➢ *Budget exhaustion* event
  - ➢ Job release → REL includes TUP
- ❑ Empirical upper bound on RUN scheduling overhead
  - ➢ $OH^{Job}_{RUN} = REL + \widehat{SCHED} + CLK + k \times (TUP + \widehat{SCHED} + max(PRE, MIG))$

$k = \lceil (3p+1)/2 \rceil$ and $\widehat{SCHED} = SCHED + CSW + LAT$.

Empirical schedulability

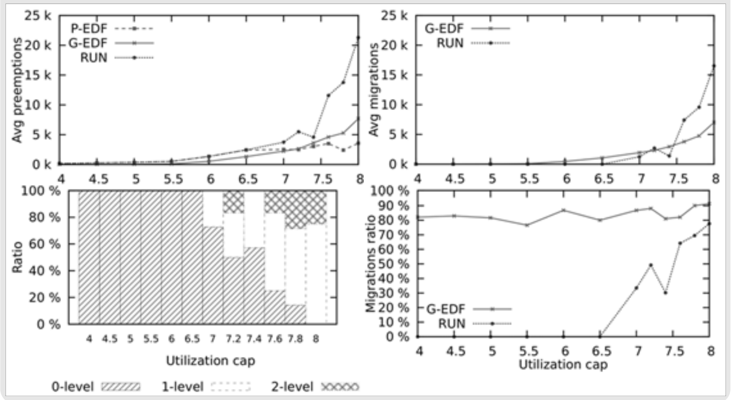




- ❑ Task sets exhibiting at least one miss
- ❑ RUN suffered **no misses**
  - ➢ Optimality and tailored overhead

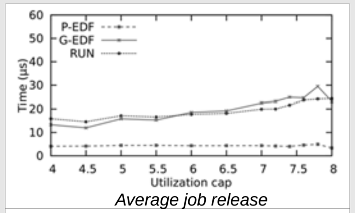
Kernel interference

- ❑ Observing average preemptions and migrations

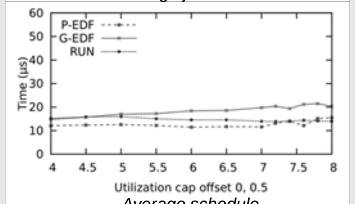


Scheduling cost

- Average cost of core scheduling primitives

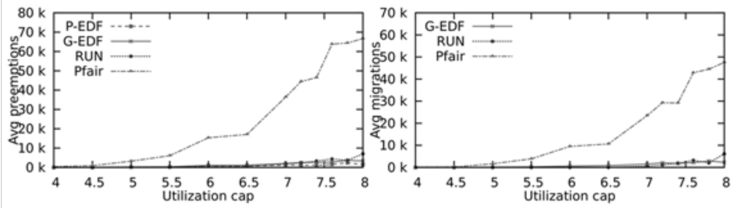


Average job release

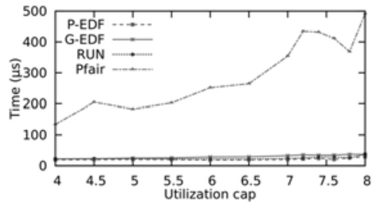


Average schedule

Evaluation against S-PD^2

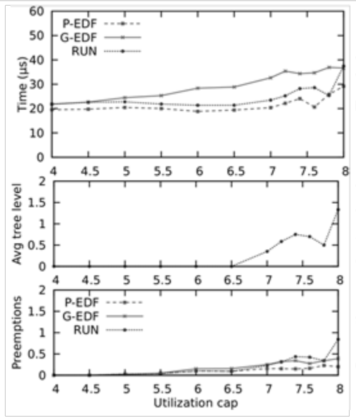


Observed preemptions and migrations

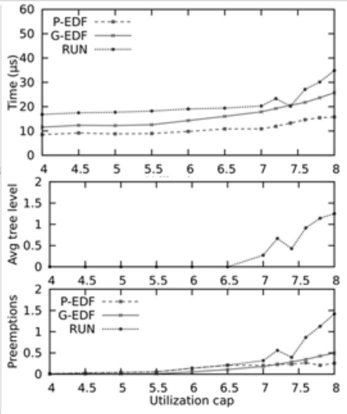


Per-job kernel overhead

Per-job scheduling overhead



Harmonic task set



Non-harmonic task set