The defeat of greedy schedulers

- Greedy algorithms are easy to explain, study, and implement
  - They work very well on single-core processors
  - EDF [1] and LLF [2] are optimal for single-core processors
- *They collapse the urgency of a job into a single value and use it to greedily schedule jobs*
- Unfortunately (and surprisingly) greedy algorithms fail when used on multiprocessors
  - EDF and LLF are no longer optimal

Why do greedy schedulers fail?

Theorem 1 (stating the obvious)

When the total utilization of a periodic task set is equal to the number of processors, then no feasible schedule can allow any processor to remain idle for any length of time

P-fair scheduling [Baruah et al. 1996]

- *Proportional progress* is a form of proportionate fairness also known as ***P-fairness***
  - Each task τ_i is assigned resources in proportion to its *weight* $W_i = \frac{C_i}{T_i}$ so that it progresses proportionately
  - Useful e.g., for real-time multimedia applications
- At every time t task τ_i must have been scheduled either $\lfloor W_i \times t \rfloor$ or $\lceil W_i \times t \rceil$ time units
  - Without loss of generality, preemption is assumed to only occur at integral time units
  - The workload model is assumed to be periodic

P-fair scheduling /2

- $\boldsymbol{lag}(S, \tau_i, t)$ is the difference between the total resource allocation that task τ_i should have received in $[0, t)$ and what it received under schedule S
- For a P-fair schedule S at time t
  - τ_i is *ahead* iff $\boldsymbol{lag}(S, \tau_i, t) < 0$
  - τ_i is *behind* iff $\boldsymbol{lag}(S, \tau_i, t) > 0$
  - τ_i is *punctual* iff $\boldsymbol{lag}(S, \tau_i, t) = 0$

P-fair scheduling /3

- $\boldsymbol{\alpha}(x)$ is the *characteristic* (infinite) *string* of task τ_x over $\{-, 0, +\}$ for $t \in \mathbb{N}$ with
  - $\boldsymbol{\alpha}_t(x) = \boldsymbol{sign}(W_x \cdot (t+1) - \lfloor W_x \cdot t \rfloor - 1)$
    - Distance from the integral approximation of fluid curve
  - $\boldsymbol{\alpha}(x, t)$ is the *characteristic substring* $\boldsymbol{\alpha}_{t+1}(x)\boldsymbol{\alpha}_{t+2}(x) \dots \boldsymbol{\alpha}_{t'}(x)$ of task τ_x at time t where $t' = min\, i: i > t: \boldsymbol{\alpha}_i(x) = 0$
- For a P-fair schedule S at time t, task τ_i is
  - *Urgent* iff τ_i is *behind* and $\boldsymbol{\alpha_t}(\tau_i) \neq -$
  - *Tnegru* iff τ_i is *ahead* and $\boldsymbol{\alpha_t}(\tau_i) \neq +$
  - *Contending* otherwise

P-fair scheduling /4

- General principle of P-fairness
  - Every task *urgent* at time t must be scheduled at t to preserve P-fairness
  - No task *tnegru* at time t can be scheduled at t without breaking P-fairness
- Problems with n_0 *tnegru*, n_1 *contending*, n_2 *urgent* tasks at time t, with m resources and $n = n_0 + n_1 + n_2$
  - If $n_2 > m$ the scheduling algorithm cannot schedule all *urgent* tasks
  - If $n_0 > n - m$ the scheduling algorithm is forced to schedule some *tnegru* tasks

Properties of a P-fair schedule S

- For task τ_i *ahead* at time t under S
  - *tnegru*: If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i not scheduled at t then τ_i is *ahead* at $t+1$
  - *tnegru*: If $\boldsymbol{\alpha_t}(\tau_i) = 0$ and τ_i not scheduled at t then τ_i is *punctual* at $t+1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i not scheduled at t then τ_i is *behind* at $t+1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i scheduled at t then τ_i is *ahead* at $t+1$
- For task τ_i *behind* at time t under S
  - If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i scheduled at t then τ_i is *ahead* at $t+1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i not scheduled at t then τ_i is *behind* at $t+1$
  - *urgent*: If $\boldsymbol{\alpha_t}(\tau_i) = 0$ and τ_i scheduled at t then τ_i is *punctual* at $t+1$
  - *urgent*: If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i scheduled at t then τ_i is *behind* at $t+1$

P-fair scheduling /5

- The **PF** scheduling algorithm
  - Schedule all *urgent* tasks
  - Allocate the remaining resources to the highest-priority *contending* tasks according to the total order function $\supseteq$ with ties broken arbitrarily
    - $x \supseteq y$ iff $\boldsymbol{\alpha}(x,t) \geq \boldsymbol{\alpha}(y,t)$
    - And the comparison between the characteristics substrings is resolved lexicographically with $- < 0 < +$
- With PF we have $\sum_{x \in [0,n]} W_x = m$
  - A dummy task may need to be added to the task set to top utilization up
- No problem situation can occur with the PF algorithm

Example (PF scheduling) /1

Task	C	T	W
v	1	3	0.333…
w	2	4	0.5
x	5	7	0.714…
y	8	11	0.727…
z	335	462	3-U

- $m = 3$ processors
- $n = 4$ tasks
- τ_z is a dummy task used to top system utilization up
- In general its period is set to the system hyperperiod
  - This time we halved it
- With PF we always have $n_2 > m$ and $n_0 \leq n - m$

8.b A stint of Deadline-Partitioning

Credits to Greg Levin et al. (ECRTS 2010)

Example (PF scheduling) /2

These tasks are scheduled and they become ahead

t	lag × period					characteristic string					urgent tasks	contending tasks	tnegru tasks
	v	w	x	y	z	v	w	x	y	z			
0	0	0	0	0	0	–	–	–	–	–	{}	$y > z > x > w > v$	{}
1	1	2	−2	−3	−127	–	0	+	+	+	$\{w\}$	$y > z > x > v$	{}
2	2	0	3	−6	−254	0	–	+	+	+	$\{v, x\}$	$w > y > z$	{}
3	0	−2	1	2	81	–	0	–	–	–	{}	$y > z > x > v$	$\{w\}$
4	1	0	−1	−1	−46	–	–	+	+	+	{}	$y > z > x > v = w$	{}
5	2	2	−3	−4	−173	0	0	+	+	+	$\{v, w\}$	$y > z > x$	{}
6	0	0	2	−7	162	–	–	0	+	+	$\{x, z\}$	$w > y > v$	{}
7	1	−2	0	1	35	–	0	–	–	–	{}	$y > z > x > v$	$\{w\}$
8	2	0	−2	−2	−92	0	–	+	+	+	$\{v\}$	$y > z > x > w$	{}
9	0	2	3	−5	−219	–	0	+	+	+	$\{w, x\}$	$y > z > v$	{}
10	1	0	1	−8	116	–	–	–	0	–	{}	$z > x > v = w$	$\{y\}$
11	−1	2	−1	0	−11	0	0	+	–	+	$\{w\}$	$y > z > x$	$\{v\}$
12	0	0	4	−3	−138	–	–	+	+	+	$\{x\}$	$y > z > w > v$	{}
13	1	2	2	−6	−265	–	0	0	+	+	$\{w, x\}$	$v > y > z$	{}
14	−1	0	0	2	70	0	–	–	–	–	{}	$y > z > x > w$	$\{v\}$
15	0	2	−2	−1	−57	–	0	+	+	+	$\{w\}$	$y > z > x > v$	{}
16	1	0	3	−4	−184	–	–	+	+	+	$\{x\}$	$y > z > v = w$	{}
17	2	2	1	−7	−311	0	0	–	+	+	$\{v, w\}$	$x > y > z$	{}
18	0	0	−1	1	24	–	–	+	–	–	{}	$y > z > x > w > v$	{}
19	1	2	−3	−2	−103	–	0	+	+	+	$\{w\}$	$y > z > v = x$	{}

Greg Levin's original presentation

- From a different deck
- The slide deck that follows proceeds from the past exam of a student of this class

DP-Fair motivation

- Focus on periodic, independent task set with implicit deadlines ($D_i = p_i$)
  - Scheduling overhead costs assumed in task requirements
  - $\sum_i U_i \leq m$ and $U_i \leq 1 \forall i$
  - Process migration allowed
- With unlimited context switches and migrations any task set meeting the above conditions will be feasible
  - This problem is easy
- What's difficult is to find a valid schedule that minimizes context switches and migrations

Deadline partitioning

- Partition time into slices demarcated by the deadlines of all tasks in the system
  - All jobs are allocated a workload in each slide and these workload share the same deadline

Theorem 2 (Hong and Leung)

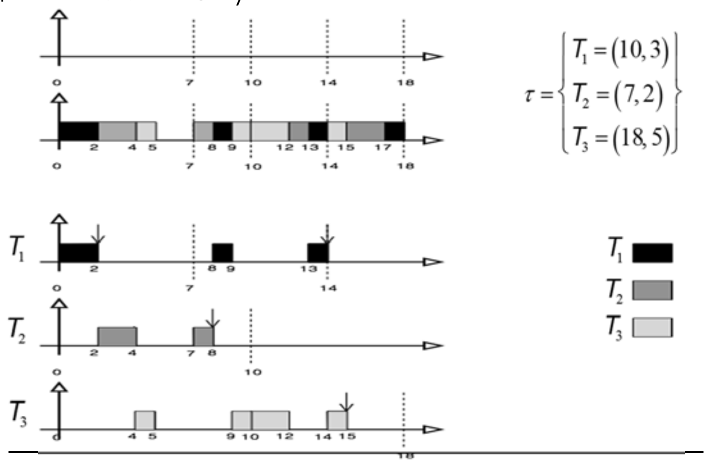
No optimal on-line scheduler can exist for a set of jobs with two or more distinct deadlines on any m multiprocessor system, where $m > 1$

- Why is DP so effective?

DP-Correct /1

- The time slice scheduler will execute all jobs' allocated workload within the end of the time slice whenever it is possible to do so
- Jobs are allocated workloads for each slice so that it is possible to complete this work within the slice
- Completion of these workloads causes all tasks' actual deadlines to be met

DP-Correct /2



Notation

- $t_0 = 0, t_i : i > 0$ denote distinct deadlines of all tasks in T
- σ_j is the j^{th} time slice in $[t_{j-1}, t_j)$
- $L_j = t_j - t_{j-1}$
- ***Local execution remaining*** $l_{i,t}$ is the amount of time that τ_i must execute before the next slice boundary
- ***Local utilization*** $r_{j,t} = l_{i,t}/(t_j - t)$
- $L_T = \sum_i l_i$ is the ***ler*** of the whole task set
- $R_T = \sum_i r_i$ is the ***lu*** of the whole task set
- *Slack* $S(T) = m - U(T)$ and represents a dummy job
- $a_{i,h}$ is the arrival time of the h^{th} job of τ_i

DP-Fair rules for periodic tasks set

- **DP-Fair allocation**
  - All tasks hit their *fluid rate curve* at the end of each slice by assigning each task a workload proportional to its utilization
  - At every σ_j assign $l_{i,t_{j-1}} = U_i \times L_j$ to τ_i
- **DP-Fair scheduling for time slices**
  - A slice-scheduling algorithm is DP-Fair if it schedules jobs within a time slice σ_i according to the following rules:
    1. Always run a job with zero local laxity
    2. Never run a job with no remaining local work
    3. Do not allow more than $S(\tau) \times L_j$ units of idle time to occur in σ_i before time t

DP-Fair optimality – Proof

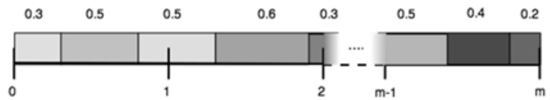
Theorem 5

Any DP-Fair scheduling algorithm for periodic task sets with implicit deadlines is optimal

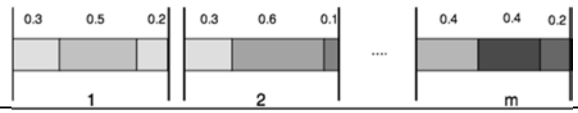
- **Lemma 3**
- If tasks in T are scheduled within a time slice by DP-Fair scheduling and $R_T \leq m$ at all times $t \in \sigma_i$, then all tasks in T will meet their local deadline at the end of the slice
- **Lemma 4**
- If a task set T of periodic tasks with implicit deadlines is scheduled in σ_i using DP-Fair algorithm, then $R_T \leq m$ will hold at all times $t \in \sigma_i$

A DP-Fair algorithm: DP-Wrap /1

- Make blocks of length δ_i for each τ_i and line these blocks up along a number line (in any order), starting at zero

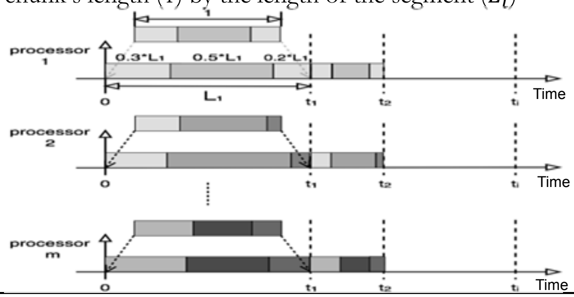


- Split this stack of blocks into chunks of length 1 at 1,2,...,m − 1



A DP-Fair algorithm: DP-Wrap /2

- Use deadline partitioning to divide time into slices
- Assign each chunk to its own processor and multiply each chunk's length (1) by the length of the segment (L_i)

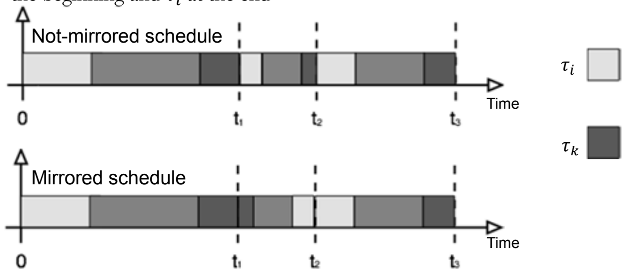


DP-Wrap features

- A very simple algorithm that satisfied all DP-Fair rules
- Almost all calculations can be done in a preprocessing step (with static task sets)
- No computational overhead at secondary events
- $n - 1$ context switches and $m - 1$ migrations per slice with *mirroring*
- Heuristics may exist to improve performance
  - Less migration and context switches

Mirroring

- For tasks that split across two slices
- If τ_i and τ_k are split and τ_i executes at the beginning and τ_k executes at the end of the slice σ_j then revert the schedule in slice σ_{j+1} so that τ_k executes at the beginning and τ_i at the end



Sporadic tasks and $D_i \leq p_i$

- DP-Fair algorithms are still optimal when $\Delta(T) \leq m$ and $\delta_i \leq 1 \; \forall i$
- Definitions
  - *Freeing slack*: unused capacity $(a_{i,h-1} + D_{i,a_{i,h}})$
  - *Active*: $(a_{i,h}, a_{j,h} + D_i)$
  - $\alpha_{i,j}(t)$, $f_{i,j}(t)$: amounts of time that task τ_i has been active or freeing slack during slice σ_j as of time t
  - *Local capacity*: $c_{i,t_{j-1}} = \delta_i \times L_i = \delta_i(\alpha_{i,j} + f_{i,j})$
  - *Freed slack* in σ_j as of time t: $F_j(t) = \sum_{i=1}^{n}(\delta_i \times f_{i,j}(t))$
  - *Slack:* $S(T) = m - \Delta(T)$

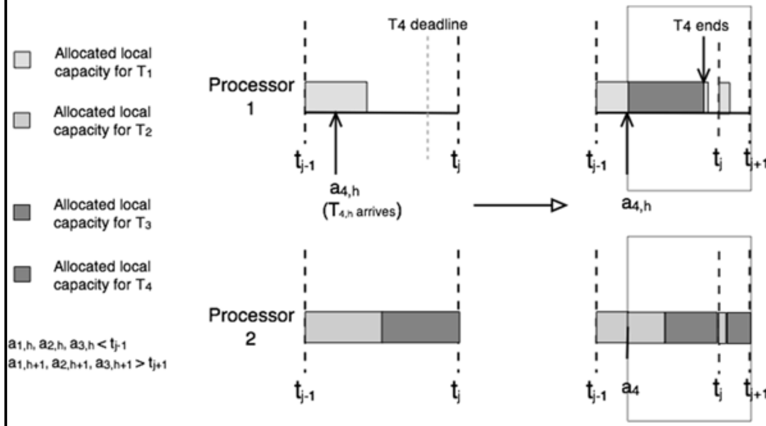
DP-Fair scheduling for time slices /1

- A slice-scheduling algorithm is DP-Fair if it schedules jobs within a time slice σ_i according to the following rules:
  1. Always run a job with zero local laxity
  2. Never run a job with no remaining local work
  3. Do not allow more than $S(T) \times L_j + F_j(t)$ units of idle time to occur in σ_i before time t
  4. Initialize $l_{i,t_{j-1}}$ to 0. At the start time t' of any active time segment for τ_i in σ_j (either $t' = t_{j-1}$ or $a_{i,h}$) that ends at time $t" = min\{a_{i,h} + D_{i,t_j}\}$, increment $l_{i,t}$ by $\delta_i(t" - t')$

DP-Fair scheduling for time slices /2

- Rules continued …
  5. When a task τ_i arrives in a slice σ_j at time t and its deadline falls within σ_j
     - Split the remainder of σ_j after t into two secondary slices σ_j^1, σ_j^2 so that the deadline of τ_i coincides with the end of σ_j^2
     - Divide the remaining local execution (and capacity) of all jobs in σ_j^1 (as well as the slack allotment from RULE 3) proportionally to the lengths of σ_j^1, σ_j^2
     - This step may be invoked recursively for any τ_k within σ_j

DP-Fair scheduling for time slices /3



Correctness

Theorem 9

Any DP-Fair scheduling algorithm is optimal for sporadic task sets with constrained deadlines where $\Delta(T) \leq m$ and $\delta_i \leq 1\ \forall i$

Proof

Lemma 7

A DP-Fair algorithm cannot cause more than $S(T) \times L_j + F_j(t)$ units of idle time in slice σ_j prior to time t

Lemma 8

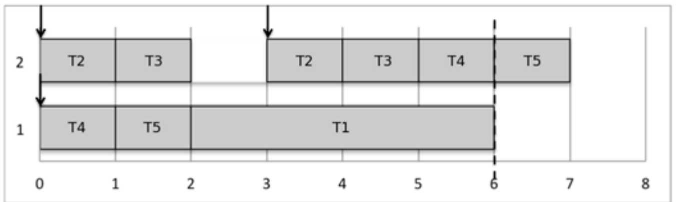
If a set T of sporadic tasks with constrained deadlines is scheduled in σ_j using a DP-Fair algorithm, then $R_t \leq m$ will hold at all times $t \in \sigma_j$

DP-Wrap modified

- If task τ_i issues a job at time t in slice σ_j and $t + D_i > t_j$ then allocate execution time $l_{i,t} = \delta_i(t_j - t)$ following RULE 4
- If instead $t + D_i < t_j$ then split the remainder of σ_j following RULE 5

Arbitrary deadlines /1

- Task set T below is <u>not</u> feasible on 2 processors
  - $m = 2, T = \{\tau_1 = (6,4), \tau_2 = \tau_3 = \tau_4 = \tau_5 = (3,1,6)\}$
  - $\Delta(T) = \frac{4}{6} + 4 \times \frac{1}{3} = 2$
  - 12 units of work to be completed by time 6



Arbitrary deadlines /2

- Is there a cure to this problem?
- If task τ_i has $D_i > p_i$ we simply impose an artificial deadline $D'_i = p_i$
- Density is not increased hence if D'_i is met, D_i will also be
- But this increases the number of context switches and migrations!

Is DP-Fair scheduling sustainable? /1

- Consider model with sporadic tasks and arbitrary deadline
- Two cases may occur
  - The new value of the relaxed parameter is not used in the scheduling and allocation policies
  - The new value of the relaxed parameter becomes known a priori/at job arrival and it is used in the scheduling and allocation policies

Is DP-Fair scheduling sustainable? /2

- Shorter execution time
  - *Case 1 (shorter* c*, same density)*
    - Task set T is schedulable and the system allocates $\delta_i \times L_j$ workload per each task in each slice
    - If $c'_i \leq c_i$ then task τ_i uses part of assigned workload and surely completes before its deadline
  - *Case 2 (shorter* c*, lesser density)*
    - As DP-Fair is optimal when $\Delta(T) \leq m$ and $\delta_i \leq 1\ \forall i = 1,..n$ a DF-Fair feasible schedule exists for T
    - A feasible schedule for T' exists as $c'_i < c_i \Rightarrow \delta'_i < \delta_i \Rightarrow \Delta(T') < D(T)$

Is DP-Fair scheduling sustainable? /3

- Longer inter-arrival time
  - *Case 1 (longer* p*, same density)*
    - Simply a less demanding instance of sporadic task
    - The allocation and scheduling rules cover this case
  - *Case 2 (longer* p*, lesser density)*
    - If $p'_i > p_i$ *and* $\delta'_i < \delta_i$ *then* $\Delta(T') < \Delta(T)$ whereby T' is feasible if T was feasible

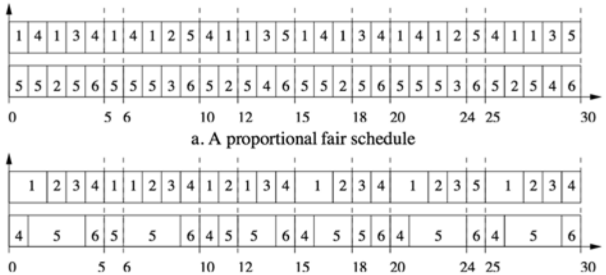
Is DP-Fair scheduling sustainable? /4

- Longer deadline
  - *Case 1 (longer* d*, same density)*
    - $d_i < d'_i$
    - Task τ'_i completes its workload at time $t = \min(d_i, p_i)$
  - *Case 2 (longer* d*, lesser density)*
    - If $d'_i > d_i$ *and* $\delta'_i < \delta_i$ *then* $\Delta(T') < \Delta(T)$ whereby T' is feasible if T was feasible
- We may therefore conclude that DP-Fair scheduling is sustainable

Related work: Boundary Fair /1

- Very similar to P-Fair
  - It still uses a function and a characteristic string to evaluate the fairness of tasks [4] with per-quantum task allocation
- It uses deadline partitioning
- It uses a less strict notion of fairness
  - At the end of every slice the absolute value of the allocation error for any task τ_i is less than one time unit
- Scheduling decisions made at the start of every slice
  - It reduces context switches packing two or more allocated time units of processor to the same task into consecutive units

Related work: Boundary Fair /2



a. A proportional fair schedule

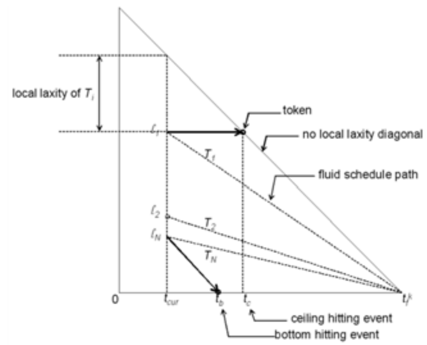
b. A boundary fair schedule

- Not DP-Fair but DP-Correct

Related work: LLREF [5] /1

- It uses deadline partitioning with DP-Wrap task allocation
- In each slice scheduling is made using the notion of T-L Plane
  - Each task T_j is represented by a token within a triangle and its position stands for the local remaining work of T_J at time i
  - The horizontal cathetus indicates the time
  - The length of the vertical cathetus is one processor's execution capacity
  - The hypotenuse represents the-no laxity line
  - Token can move in two directions. Horizontally if the task doesn't execute, diagonally down if it does
  - When a token hits the horizontal cathetus or the hypotenuse (secondary events) a scheduling decision is made
    - Tasks are sorted and m tasks with the least laxity are executed

Related work: LLREF /2



- DP-Fair algorithm but does unnecessary work

Useful DP-Fair bibliography

1. C. Liu and J. Layland, "Scheduling Algorithms for Multi-programming in a Hard-Real-Time Environment", Journal of the ACM (JACM), 20(1):46–61, 1973
2. A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment", Technical report, Massachusetts Institute of Technology, 1983
3. S. K. Cho, S. Lee, A. Han, and K.-J. Lin, "Efficient Real- Time Scheduling Algorithms for Multiprocessor Systems", IEICE Transactions on Communications, E85-B(12):2859– 2867, 2002
4. D. Zhu, D. Mossé and R. Melhem, "Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary?", IEEE Real-Time Systems Symposium (RTSS), 2003
5. H. Cho, B. Ravindran and E. Jensen, "An Optimal Real-Time Scheduling Algorithm for Multiprocessors", IEEE Real-Time Systems Symposium (RTSS), 2006
6. B. Andersson and, E. Tovar, "Multiprocessor Scheduling with Few Preemptions", IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA), 2006
7. K. Funaoka, S. Kato and N. Yamasaki, "Work-Conserving Optimal Real-Time Scheduling on Multiprocessors" Euromicro Conference on Real-Time Systems (ECRTS), 2008
8. S. Funk and V. Nadadur "LRE-TL: An Optimal Multiprocessor Algorithm for Sporadic Task Sets", Conference on Real-Time and Networked Systems (RTNS), 2009

Related work: EKG [6]

- Tasks are divided into heavy and light
  - Each heavy task is assigned to a dedicate processor
  - Every light task is assigned to one group of K processors and it shares them with other light tasks
- Some light tasks are split in two processors and they are executed either before t_a or after t_b
- Light tasks that are not split are executed between t_a or and t_b and they are scheduled by EDF
- Heavy tasks start executing when they become ready
- EDF is not a DP-Fair allocation but the DP-Fair rules are satisfied

8.c More theoretical results

More theoretical results /1

- For the simplest workload model made of independent periodic and sporadic tasks
- A *P-fair* scheme can sustain $U = m$ for m processors but its run-time overheads are excessive
  - Tasks incur very many preemptions and are frequently required to migrate → *horrendously costly disruption*
- *Partitioned FPS first-fit* (on decreasing task utilization) can sustain $U \leq m(\sqrt{2} - 1)$
  - But this is a sufficient test only [Oh & Baker, 1998]

More theoretical results /2

- *Partitioned EDF first-fit* can sustain

$$U \leq \frac{\beta m + 1}{\beta + 1}$$

$$\beta = \left\lfloor \frac{1}{U_{\max}} \right\rfloor$$

Per task

- For high U_{max} this bound gets rapidly lower than $0.6 \times m$, but can get close to m for some examples
  - Again this is a sufficient test only [Lopez *et al.*, 2004]

More theoretical results /3

- *Global EDF* can sustain

$$U \leq m - (m-1)U_{\max}$$

- For high U_{max} this bound can be as low as $0.2 \times m$ but also close to m for other examples
  - Again, only sufficient [Goossens *et al.*, 2003]

More theoretical results /4

- Combinations
  - FPS (higher band) to those tasks with $U_i > 0.5$
  - EDF for the rest

$$U \leq \left(\frac{m+1}{2}\right)$$

  - Again, only sufficient [Baruah, 2004]