Embedded Systems 20

Multiprocessor Scheduling

REVIEW

Given

- n equivalent processors,
- a finite set M of aperiodic/periodic tasks

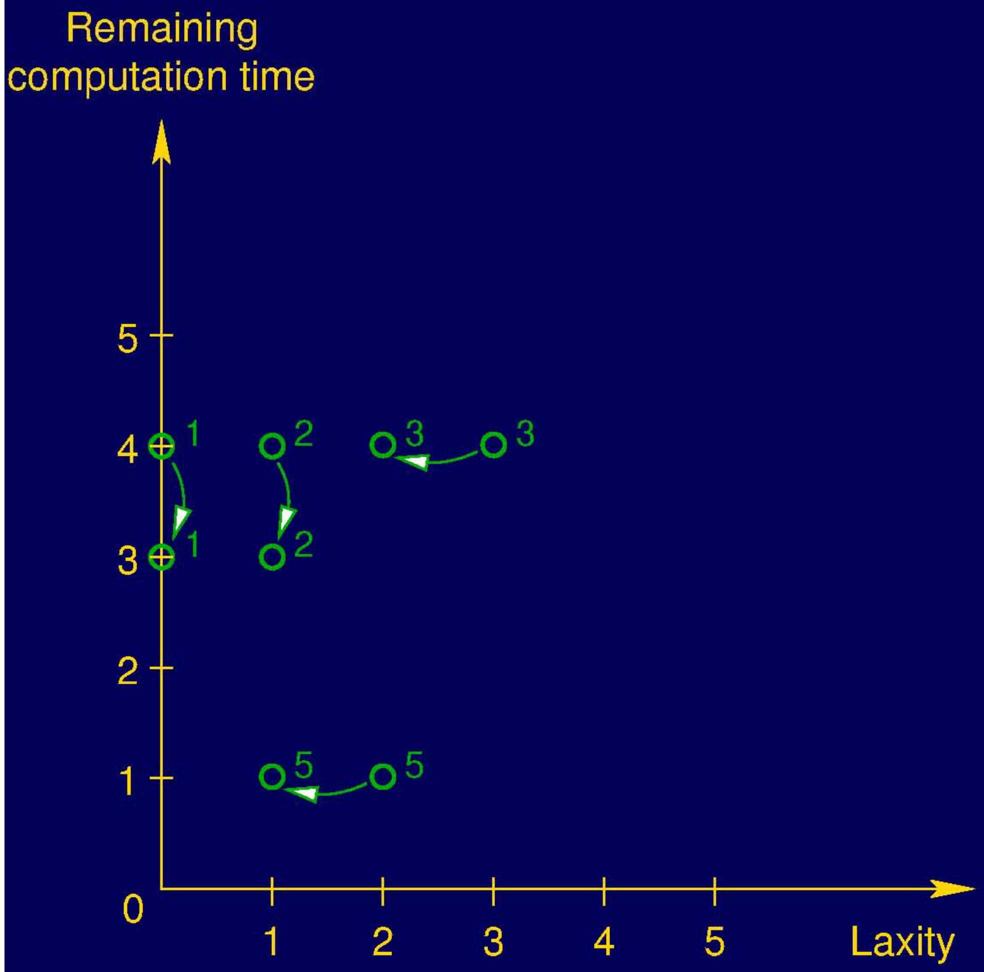
find a schedule such that each task always meets its deadline.

Assumptions:

- Tasks can freely be migrated between processors
  - at any integer time instant, without overhead
  - however: no task may run on two processors simultaneously
- All tasks are preemptable
  - at any integer time instant, without overhead

Game-board representation

REVIEW



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

Nodes reaching the x–axis have been allocated all the computation time they need and are thus removed from the game board, as they don't represent scheduling constraints any longer.

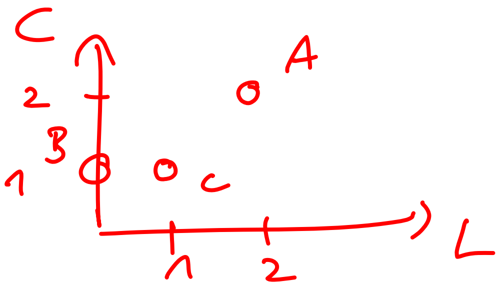
The game is lost (i.e., the schedule is infeasible) if some node reaches the second quadrant.

It is won if no node remains on the board.

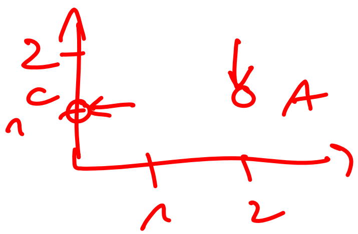
Online scheduling?

REVIEW

Theorem: There can be no optimal scheduling algorithm if the release times are not known a priori.



Case 1: Scheduler selects A + B



New tasks D, E with L = 0, C = 1
arrive at t = 1
=> no schedule

Feasible schedule: B + C, D + E, A

Game-theoretic solution

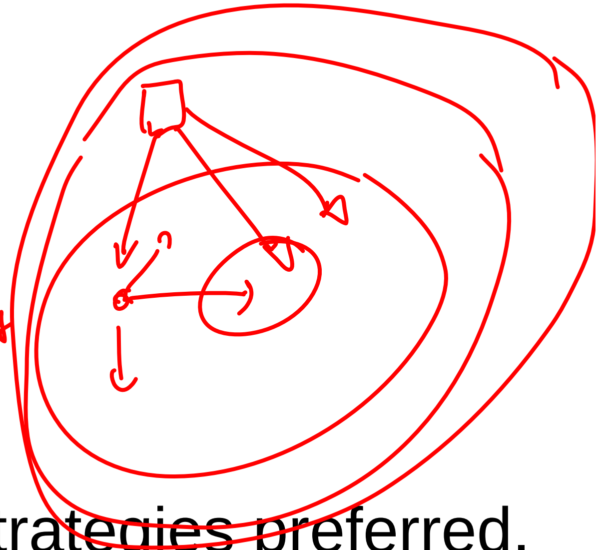
REVIEW

Theorem: In games with

- **finitely many positions** on the game board, and
- **complete** information

there is a always a winning strategy for one of the two players; it can be constructed effectively.

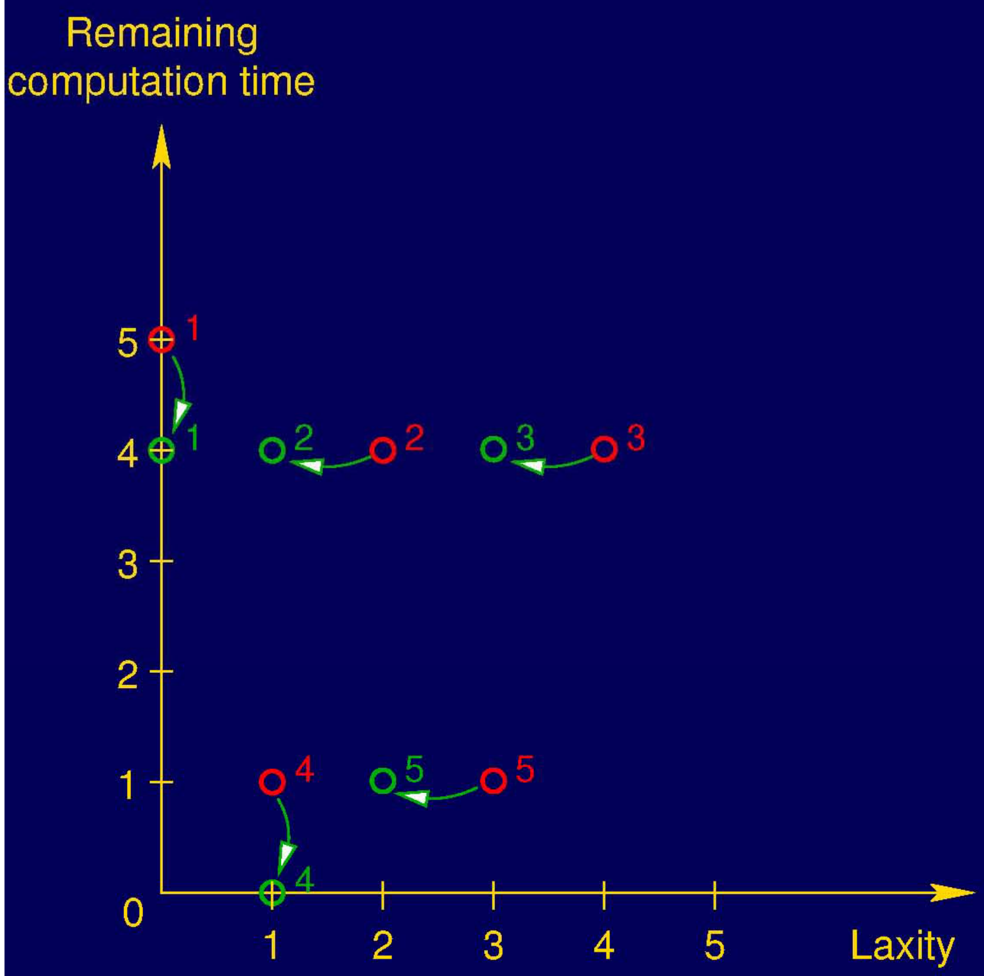
Fixpoint construction:
- Start with the winning position
- Add all positions where we can move into set
- Add all positions where the opponent must move into set
- repeat until no more change



However: high complexity $\Rightarrow$ predefined strategies preferred.

LLF (Least Laxity First)

REVIEW



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

In every time scheduling step / turn of the game:

– at most n nodes go down by 1

– the rest moves 1 to the left

LLF is optimal.

Periodic ~~periodic~~ tasks **REVIEW**

Theorem: A necessary and sufficient condition for the schedulability of periodic tasks is that $\underline{U \leq n}$.

necessary √ .

Scheduling idea

REVIEW

1. Divide the time line into time slices such that each period of each process is divided into an integral number of time slices.
   Slice length $T = \text{GCD}(T_1, \ldots, T_n)$.
2. Within each time slice, allocate processor time in proportion to the utilization $U_i = \frac{C_i}{T_i}$ originating from the various tasks.
   Processing time per slice $r_i = TU_i = T\frac{C_i}{T_i}$.
   Hence, each task runs $\frac{T_i}{T} r_i = \frac{T_i}{T} T \frac{C_i}{T_i} = C_i$ time units within its period.
3. Allocate r_i according to the following algorithm
   (a) Look for the first processor proc_j that has free capacity in its time slices.
   (b) Allocate that portion of r_i to proc_j that proc_j can accommodate.
   (c) If all of r_i has been allocated then proceed with the next task (goto step a).
   (d) Otherwise allocate the remainder of r_i to proc_{j+1}.
   proc_{j+1} has enough spare capacity as it has not previously been used and $r_i \leq T$ due to $U_i \leq 1$. Furthermore, due to $r_i \leq T$, we don't generate temporal overlap between the two partial runs of task i.

Example (2 processors)

i	C_i	T_i
1	2	4
2	8	8
3	3	6

$$U = \frac{2}{4} + \frac{8}{8} + \frac{3}{6} = 2$$

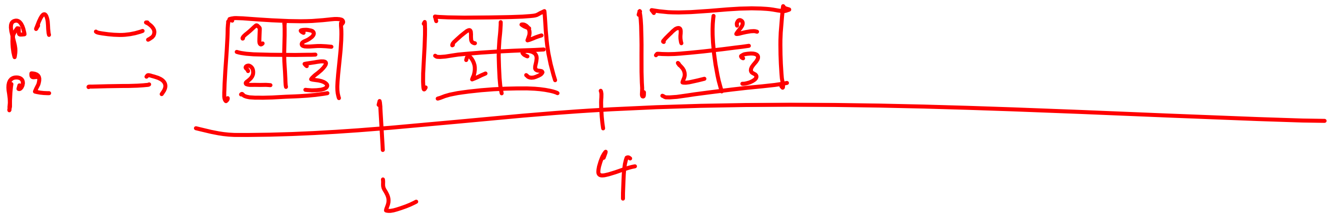
$$T = \gcd(4, 8, 6) = 2$$

In each slice,

task$_1$ has $2 \cdot \frac{2}{4} = 1$ unit

task$_2$ has $2 \cdot \frac{8}{8} = 2$ units

task$_3$ has $2 \cdot \frac{3}{6} = 1$ unit



Scheduling idea

This scheme works if

- the load isn't too high:

$$U = \sum_{i \in M} \frac{C_i}{T_i} \leq n$$

  and

- the time slices allocated have integral length:

$$r_i = TU_i = T\frac{C_i}{T_i} \in \mathbb{N} \text{ for each } i \in M$$

Rescheduling fractional parts

- Let $X_i = T*C_i/T_i - \lfloor T*C_i/T_i \rfloor$
- In each period,
  allocate in $X_i * T_i/T$ slices: $\lfloor T*C_i/T_i \rfloor + 1$ units
  and in all other slices: $\lfloor T*C_i/T_i \rfloor$ units
- This can be done without allowing any task to miss its deadline: use EDF!

Example (2 processors)

i	C_i	T_i
1	2	4
2	4	6
3	3	~~6~~ 4

$U = \frac{2}{4} + \frac{4}{6} + \frac{3}{\cancel{6}\,4} < 2$

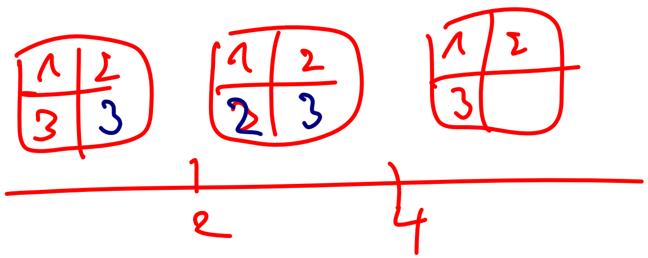
$T = gcd(4,6,6) = 2$

In each slice,

task$_1$ has $\lfloor 2 * \frac{2}{4} \rfloor = 1$ time unit

task$_2$ has $\lfloor 2 * \frac{4}{6} \rfloor = 1$ time unit

task$_3$ has $\lfloor 2 * \frac{3}{\cancel{6}\,4} \rfloor = 1$ time unit



task$_2$ and task$_3$ have fractional parts

$\Rightarrow$ task$_2$ needs 1 more unit every 3 slices; task$_3$ needs 1 more unit every 2 slices

Extension: Task migration time

Theorem: A **necessary** and **sufficient** condition for scheduling periodic tasks on n processors is
$U \leq n$,
if the task migration time is one unit.

Proof: Induction over length of schedule.

Extension: Task migration time

Lemma: If U ≤ n, then **within each time slice** the tasks can meet the migration time requirement without missing deadlines, if the task migration time is one unit.

- Sort tasks according to non-increasing computation blocks
- If computation block = T
  → allocate processor exclusively
- If computation block < T
  → allocate part of computation block at end of proc i
  and part at beginning of proc i+1
  ⇒ gap of at least 1 unit; or:
  → allocate entire computation block
  → no migration.

Extension: Task migration time

Lemma: If U ≤ n, then **between time slices** the tasks can meet the migration time requirement without missing deadlines, if the task migration time is one unit.

- For each time slice, sort tasks according to their increasing computation slacks
- If computation slack = T → find processor that executes task at end of the previous slice
  ⇒ no migration
- No such processor ⇒ assign to same left-over processor in the end (migration time accounted for in previous slice).
- If computation slack < T → find processor j that executed task at end of previous slice
  → assign as much as possible to current processor

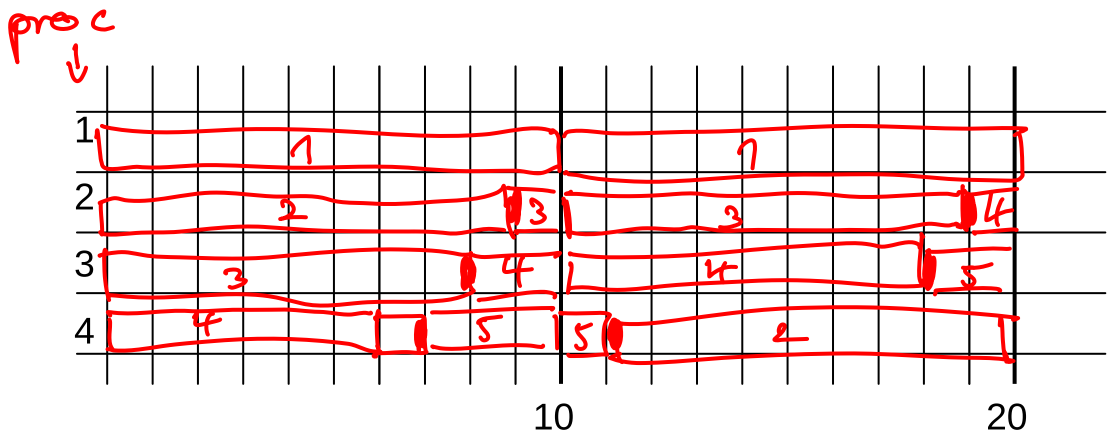
if insufficient → use processor if free beginning.
(no migration at beginning; within slice ≥ 1
unit migration time).

no such processor ⇒ assign later
(migration time accounted for in previous
slice).

Example (4 processors)

i	*Computation block*
1	10
2	9
3	9
4	9
5	3

T=10



Extension: Task migration time

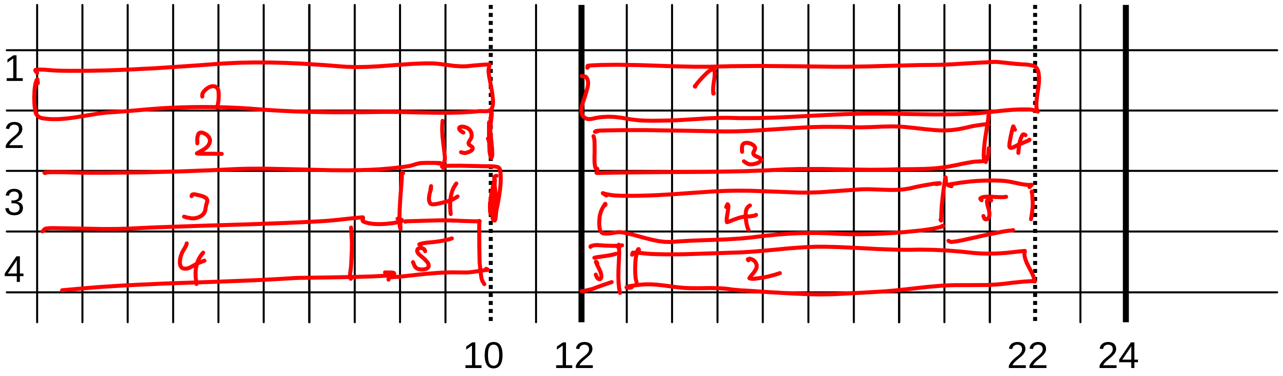
Theorem: Let $T = \gcd(T_1, \ldots, T_m)$ and let R be~~t~~ the task migration time. A **sufficient condition** for scheduling the m periodic tasks is that $U \leq n \cdot (T-R+1)/T$.

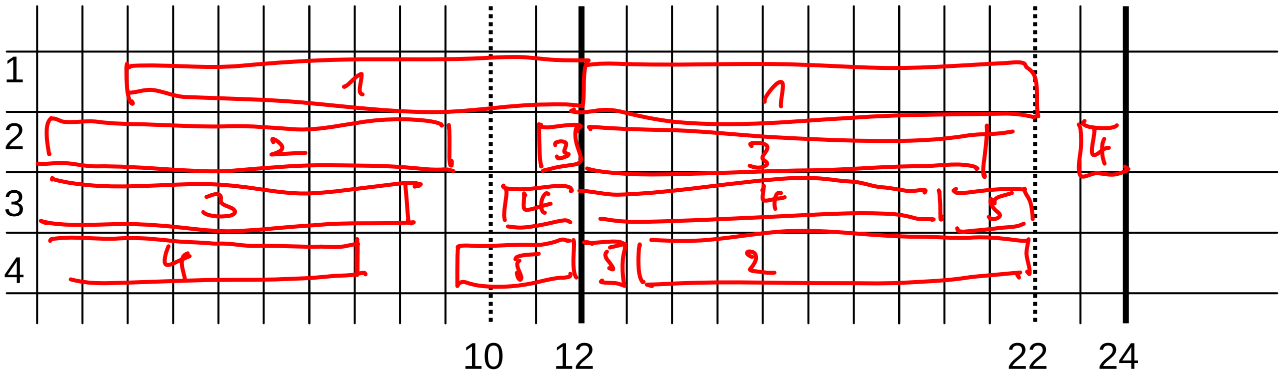
- Schedule as before, but use only initial $T-R+1$ units of slice
- Whenever migration time too short
  → shift task(s) to right end of slice.

Example (4 processors)

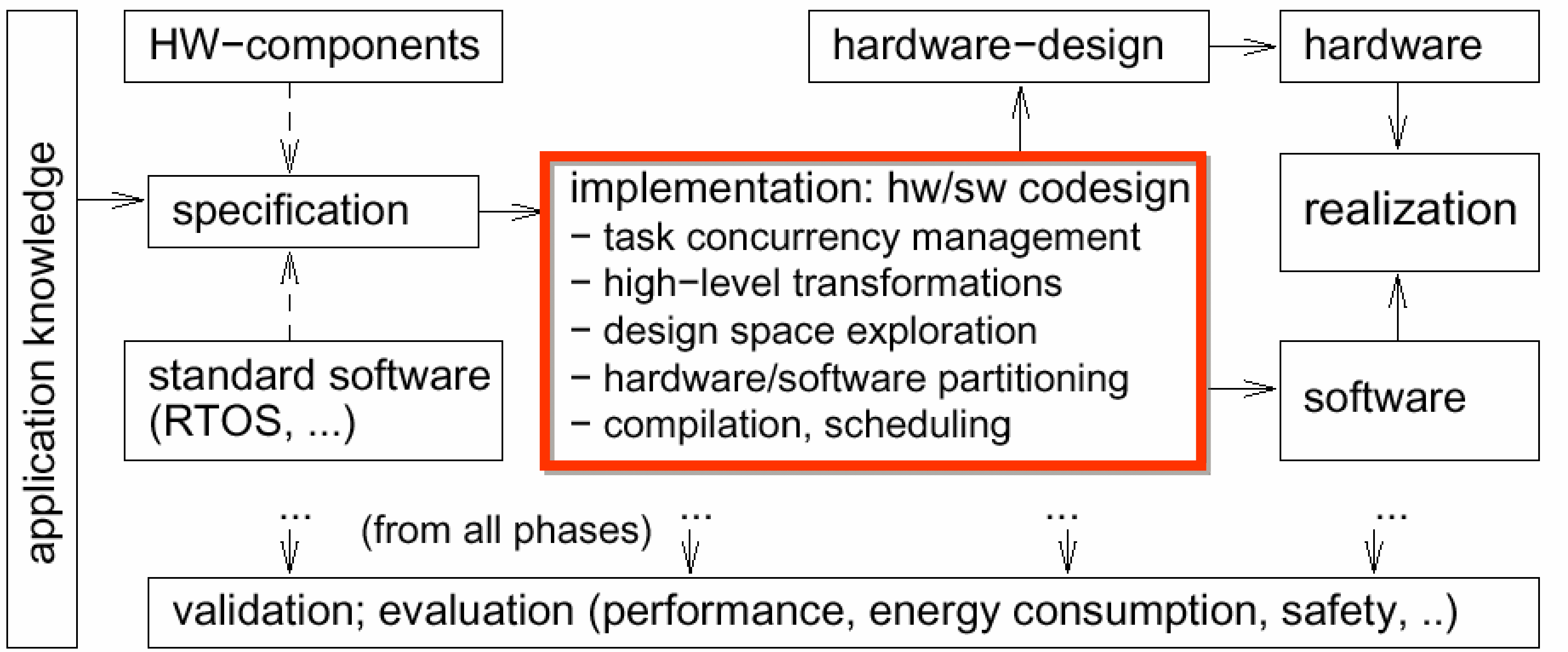
i	*Computation block*
1	10
2	9
3	9
4	9
5	3

T=12,
R=3





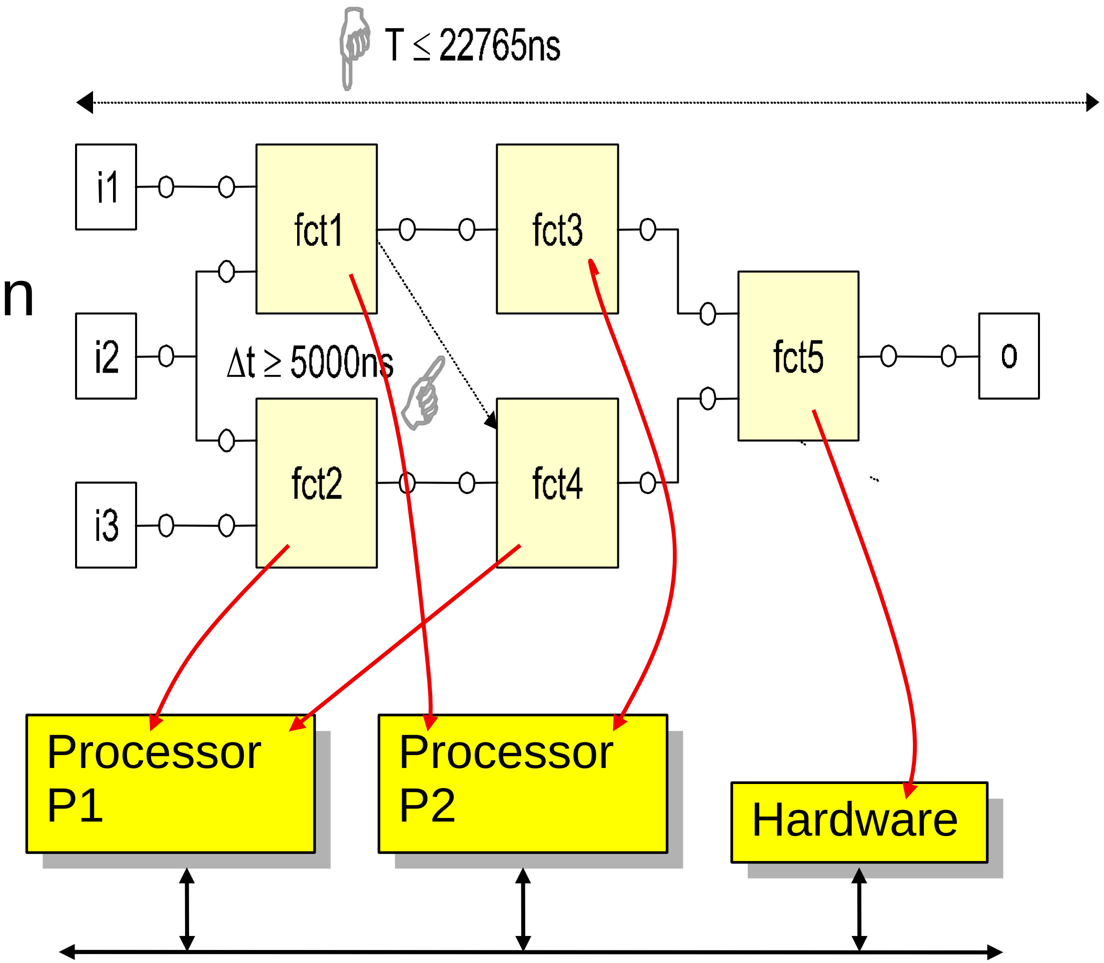
Overview


application knowledge
HW−components
specification
standard software (RTOS, ...)
hardware−design
hardware
implementation: hw/sw codesign
− task concurrency management
− high−level transformations
− design space exploration
− hardware/software partitioning
− compilation, scheduling
realization
software
...
(from all phases)
...
...
...
validation; evaluation (performance, energy consumption, safety, ..)

Hardware/software codesign

Specification

Mapping



The Partitioning Problem

Definition: The **partitioning problem** is to assign
n **objects** $O=\{o_1, \ldots, o_n\}$ to
m **blocks** (also called **partitions**) $P=\{p_1, \ldots, p_m\}$
such that

- $p_1 \cup p_2 \ldots \cup p_m = O$
- $p_i \cap p_j = \varnothing$ for all $i \neq j$, and
- cost $c(P)$ is minimized.

Cost function (Estimated) quality of design, may include

- System price
- Latency
- Power consumption, …

Partitioning Methods

- Exact methods
  - Enumeration
  - Integer Linear Programming (ILP)
- Heuristic methods
  - Constructive methods
    - Random mapping
    - Hierarchical clustering
  - Iterative methods
    - Kernighan-Lin Algorithm
    - Simulated Annealing
    - …

Integer programming models

- Ingredients:
- Cost function
- Constraints

$\left.\begin{array}{l}\text{Cost function}\\ \text{Constraints}\end{array}\right\}$ Involving linear expressions over *integer* variables from a set X

Cost function $$C = \sum_{x_i \in X} a_i x_i \text{ with } a_i \in R, x_i \in \mathbb{N} \quad (1)$$

Constraints: $$\forall j \in J : \sum_{x_i \in X} b_{i,j}\, x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \quad (2)$$

Def.: The problem of minimizing (1) subject to the constraints (2) is called an **integer programming (IP) problem**.

If all x_i are constrained to be either 0 or 1, the IP problem said to be a **0/1 integer programming problem**.

Example

$$C = 5x_1 + 6x_2 + 4x_3$$

$$x_1 + x_2 + x_3 \geq 2$$

$$x_1, x_2, x_3 \in \{0,1\}$$

x_1	x_2	x_3	C
0	1	1	10
1	0	1	9
1	1	0	11
1	1	1	15

Optimal

Remarks on integer programming

- Integer programming is NP-complete.
- Running times depend exponentially on problem size, but problems of >1000 vars solvable with good solver (depending on the size and structure of the problem)
- The case of $x_i \in \mathbb{R}$ is called *linear programming* (LP). LP has polynomial complexity, but most algorithms are exponential, still in practice faster than for ILP problems.
- The case of some $x_i \in \mathbb{R}$ and some $x_i \in \mathbb{N}$ is called *mixed integer-linear programming.*
- ILP/LP models can be a good starting point for modeling, even if in the end heuristics have to be used to solve them.

Integer Linear Programming for Partitioning

- Binary variables $x_{i,k}$
  - $x_{i,k}$=1: object o_i in block p_k
  - $x_{i,k}$=0: object o_i not in block p_k
- Cost $c_{i,k}$ if object o_i in block p_k
- Integer linear program:

$$x_{i,k} \in \{0,1\} \qquad 1 \le i \le n, 1 \le k \le m$$

$$\sum_{k=1}^{m} x_{i,k} = 1 \qquad 1 \le i \le n$$

minimize $\displaystyle\sum_{k=1}^{m} \sum_{i=1}^{n} x_{i,k} \cdot c_{i,k}$

Extensions

- Constraints:
  Example: maximum number of objects in block: h_k

$$\sum_{i=1}^{n} x_{i,k} \leq h_k$$

- Scheduling
- Component types
- Different costs (processor/memory/ASIC)

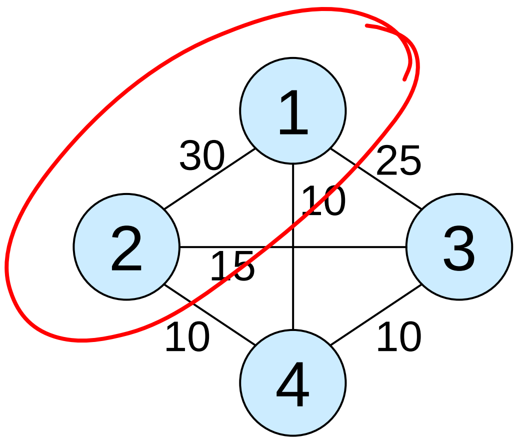
Constructive Methods

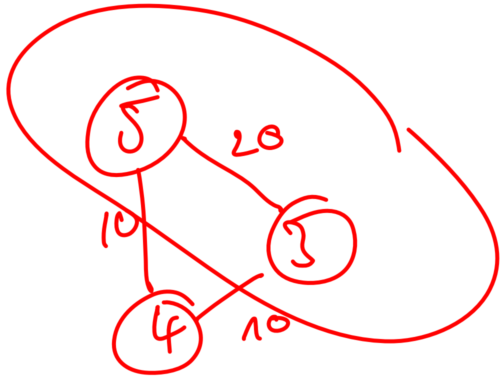
- **Random mapping**
  - Each object randomly assigned to some block
  - Used to find starting partition for iterative methods
- **Hierarchical clustering**
  - Assumes closeness function: determines how desirable it is to group two objects
  - Start with singleton blocks
  - Repeat until termination criterion (e.g., desired number of blocks reached)
    - Compute closeness of blocks (average closeness of object pairs)
    - Find pair of closest blocks
    - Merge blocks
  - Difficulty: find proper closeness function

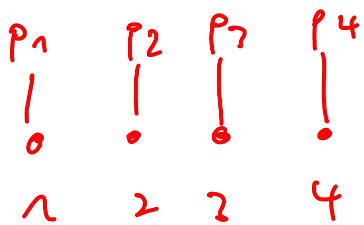
Example: Hierarchical Clustering

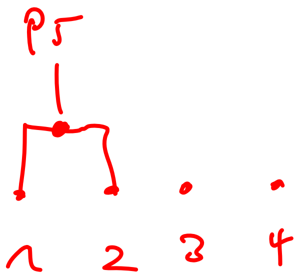
Average closeness;

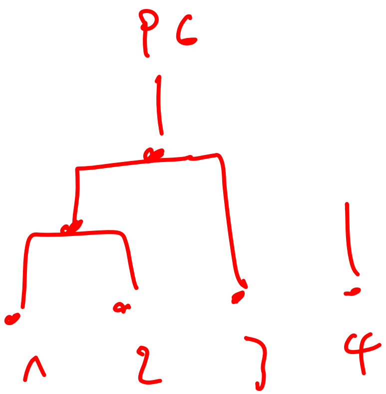
Termination: 2 blocks

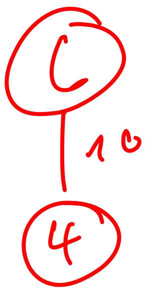












Ratiocut

$$ratio = \frac{cut(P)}{size(p_i) \cdot size(p_j)}$$

where

- $P = \{p_i, p_j\}$
- cut(P)= sum of closeness between elements in p_i and p_j

Hw/Sw Partitioning

- Special case: Bi-partitioning $P=\{p_{SW}, p_{HW}\}$
- Software-oriented approach: $P=\{O,\varnothing\}$
  - In software, all functions can be realized
  - Performance might be too low $\Rightarrow$ migrate objects to HW
- Hardware-oriented approach: $P=\{\varnothing,O\}$
  - In hardware, performance is OK
  - Cost might be too high $\Rightarrow$ migrate objects to SW

Greedy Hw/Sw Partitioning

Migration of objects to the other block (HW/SW) until no more improvement

```
repeat
  begin
  P'=P;
  for i=1 to n
      begin
       if (cost(move(P,o_i) < cost(P))
       then P':=move(P,o_i);
      end;
 end;
until (P==P')
```