Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems

Institute of Communication, Information and Perception Technologies
Scuola Superiore Sant'Anna
Retis Real-Time Systems Laboratory

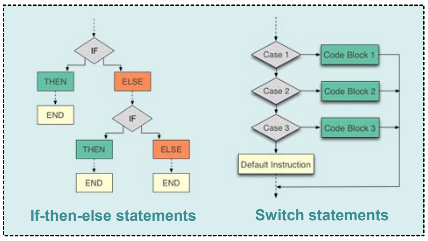
Alessandra Melani

What does it mean?

- « Response-time analysis » ✓
- « conditional »
- « DAG tasks »
- « multiprocessor systems » ✓

What does it mean?

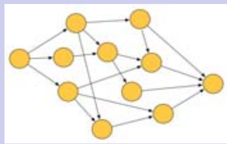
- « Response-time analysis »
- **« conditional »**
- « DAG tasks »
- « multiprocessor systems »



What does it mean?

- « Response-time analysis »
- « conditional »
- **« DAG tasks »**
- « multiprocessor systems »

DAG: Directed Acyclic Graph



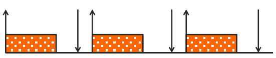
In other words

- We will analyze a **multiprocessor** real-time systems…
- … by means of a **schedulability test** based on **response-time analysis**
- … assuming **Global Fixed Priority** or **Global EDF** scheduling policies
- … and assuming a **parallel task model** (i.e., a task is modelled as a **Directed Acyclic Graph - DAG**)

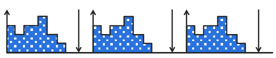
Parallel task models

Many parallel programming models have been proposed to support parallel computation on multiprocessor platforms (e.g., OpenMP, OpenCL, Cilk, Cilk Plus, Intel TBB)

Early real-time scheduling models: each recurrent task is completely sequential

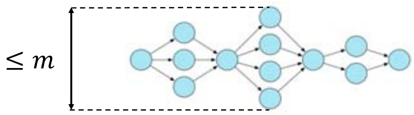


Recently, more expressive execution models allow exploiting task parallelism



Fork-join

- Each task is an alternating sequence of sequential and parallel segments
- Every parallel segment has a degree of parallelism $\leq m$ (number of processors)

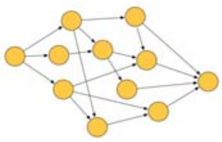


Synchronous-parallel

- Generalization of the fork-join model
- Allows consecutive parallel segments
- Allows an arbitrary degree of parallelism of every segment
- Synchronization at segment boundaries: a sub-task in the new segment may start only after completion of all sub-tasks in the previous segment

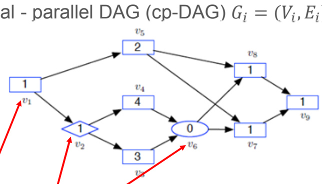
DAG

- Directed acyclic graph (DAG) $G_i = (V_i, E_i)$
- $V_i = \{v_{i,1}, \ldots, v_{i,n_i}\}$; $E_i \subseteq V_i \times V_i$
- Generalization of the previous two models
- Every node is a sequential sub-task
- Arcs represent precedence constraints between sub-tasks



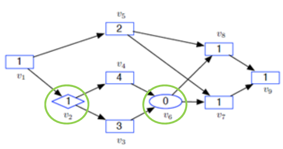
cp-DAG

- Conditional - parallel DAG (cp-DAG) $G_i = (V_i, E_i)$



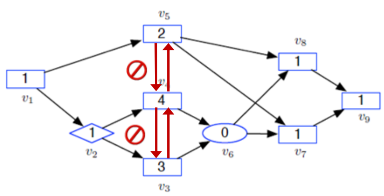
- Two types of nodes
  - **Regular**: all successors must be executed in parallel
  - **Conditional**: to model start/end of a conditional construct (e.g., if-then-else statement)
- Each node has a WCET $C_{i,j}$
- In this lecture, we will focus on **this** task model

Conditional pairs



- (v_2, v_6) form a **conditional pair**
  - v_2 is a starting conditional node
  - v_6 is the joining point of the conditional branches starting at v_2
- **Restriction**: there cannot be any connection between a node belonging to a branch of a conditional statement (e.g., v_4) and nodes outside that branch (e.g., v_5), including other branches of the same statement

Why this restriction?

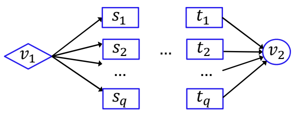


- It does not make sense for v_5 to wait for v_4 if v_3 is executed
- Analogously, v_4 cannot be connected to v_3 since only one is executed
- Violation of the correctness of conditional constructs and the semantics of the precedence relation

Formal definition (1)

Let (v_1, v_2) be a pair of conditional nodes in a DAG $G_i = (V_i, E_i)$.
The pair (v_1, v_2) is a conditional pair if the following hold:

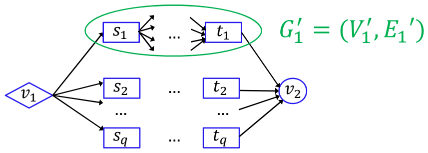
- Suppose there are exactly q outgoing arcs from v_1 to the nodes $s_1, s_2, \ldots, s_q$, for some $q > 1$. Then there are exactly q incoming arcs into v_2 in E_i, from some nodes $t_1, t_2, \ldots, t_q$



Formal definition (2)

- For each $l \in \{1,2,\ldots,q\}$, let $V_l' \subseteq V_i$ and $E_l' \subseteq E_i$ denote all the nodes and arcs on paths reachable from s_l that do not include v_2.

  By definition, s_l is the sole source node of the DAG $G_l' = (V_l', E_l')$. It must hold that t_l is the sole sink node of G_l'.

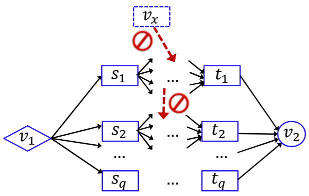


Formal definition (3)

- It must hold that $V_l' \cap V_j' = \emptyset$ for all $l, j, l \neq j$.

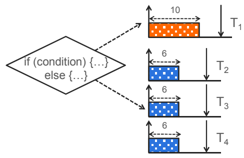
  Additionally, with the exception of (v_1, s_l), there should be no arcs in E_i into nodes in V_l' from nodes not in V_l', for each $l \in \{1,2,\ldots,q\}$.

  That is, $E_i \cap \left((V_i \backslash V_l') \times V_l'\right) = \{(v_1, s_l)\}$ should hold for all l.



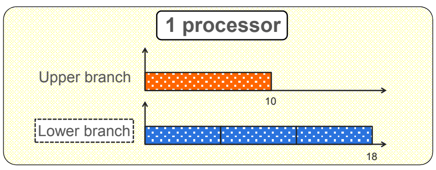
How is parallel code structured?

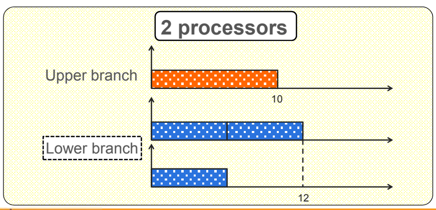
```
#pragma omp parallel num_threads(N)
{
  #pragma omp master {
  #pragma omp task { // T0
       if (condition) {
          #pragma omp task { // T1 }
       } else {
          #pragma omp task { // T2 }
          #pragma omp task { // T3 }
          #pragma omp task { // T4 }
       }
}}}
```

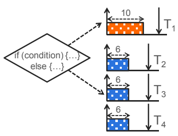


Which branch leads to the worst-case response-time?

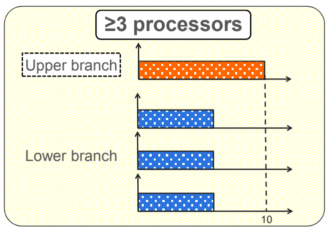
Which branch leads to the WCRT?

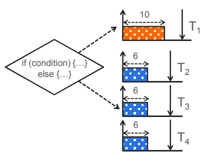


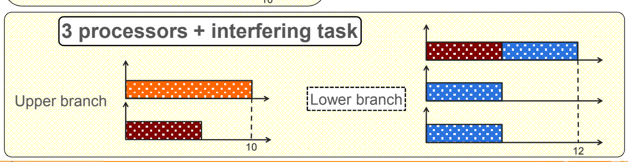




Which branch leads to the WCRT?







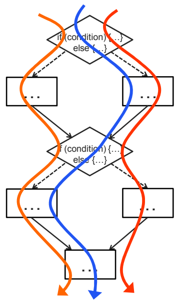
Lesson learnt

Depending on the number of processors and on the interfering tasks, it is not obvious to identify the **branch leading to the WCRT**

It makes sense to account for the different execution flows by **enriching the task model**

Why don't we do it also with **sequential** tasks?

- Only the longest path matters
- Conditional branches are already *incorporated in the notion of WCET*



System model

- n conditional-parallel tasks (cp-tasks) τ_i, expressed as cp-DAGs in the form $G_i = (V_i, E_i)$
- platform composed of m identical processors
- **sporadic** arrival pattern (minimum inter-arrival time T_i between jobs of task τ_i)
- **constrained** relative deadline $D_i \le T_i$

Problem

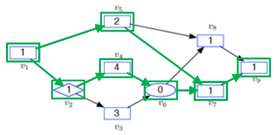
Schedulability analysis for cp-tasks, **globally** scheduled on **m identical processors with any work-conserving algorithm (including G-FP and G-EDF)**

Quantities of interest

1. Chain (or path) of a cp-task
2. Longest path
3. Volume
4. Worst-case workload
5. Critical chain

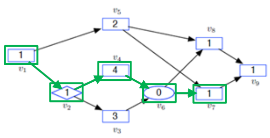
1. Chain (or path)

A chain (or path) of a cp-task τ_i is a sequence of nodes $\lambda = (v_{i,a}, \dots, v_{i,b})$ such that $(v_{i,j}, v_{i,j+1}) \in E_i, \forall j \in [a, b)$.



1. Chain (or path)

A chain (or path) of a cp-task τ_i is a sequence of nodes $\lambda = (v_{i,a}, \dots, v_{i,b})$ such that $(v_{i,j}, v_{i,j+1}) \in E_i, \forall j \in [a, b)$.

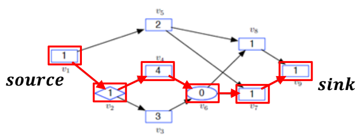


The length of the chain, denoted by $len(\lambda)$, is the sum of the WCETs of all its nodes:

$$len(\lambda) = \sum_{j=a}^{b} C_{i,j}$$

2. Longest path

The longest path L_i of a cp-task τ_i is any source-sink chain of the task that achieves the longest length



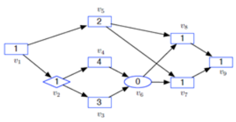
L_i also represents the time required to execute it when the number of processing units is infinite (large enough to allow maximum parallelism)

Necessary condition for feasibility: $L_i \le D_i$

2. Longest path

How to compute the longest path?

1. Find a topological order of the given cp-DAG
   - A topological order is such that of there is an arc from u to v in the cp-DAG, then u appears before v in the topological order → can be done in $O(n)$
   - Example: for this cp-DAG possible topological orders are
     - $(v_1, v_2, v_5, v_3, v_4, v_6, v_8, v_7, v_9)$
     - $(v_1, v_5, v_2, v_3, v_4, v_6, v_7, v_8, v_9)$
     - $(v_1, v_2, v_4, v_3, v_6, v_5, v_8, v_7, v_9)$



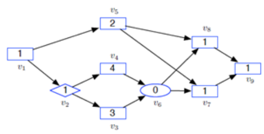
2. Longest path

How to compute the longest path?

2. For each vertex $v_{i,j}$ of the cp-DAG in the topological order, compute the length of the longest path ending at $v_{i,j}$ by looking at its incoming neighbors and adding $C_{i,j}$ to the maximum length recorded for those neighbors

   If $v_{i,j}$ has no incoming neighbors, set the length of the longest path ending at $v_{i,j}$ to $C_{i,j}$

   Example:
   - For v_1, record 1
   - For v_2, record 2
   - For v_3, record 5
   - For v_4, record 6
   - For v_5, record $\max(5, 6) = 6$

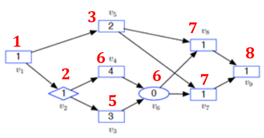


2. Longest path

How to compute the longest path?

3. Finally, the longest path in the cp-DAG may be obtained by starting at the vertex $v_{i,j}$ with the largest recorded value, then repeatedly stepping backwards to its incoming neighbor with the largest recorded value, and reversing the sequence found in this way

   Example: **recorded values**



- Starting at v_9 and stepping backward we find the sequence $(v_9, v_7, v_6, v_4, v_2, v_1)$
- The longest path is then $(v_1, v_2, v_4, v_6, v_7, v_9)$

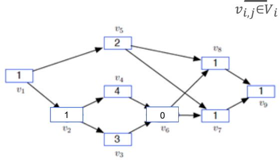
Complexity of the longest path computation: $O(n)$

3. Volume

In the **absence** of conditional branches, the volume of a task is the worst-case execution time needed to complete it on a dedicated single-core platform

It can be computed as the sum of the WCETs of all its vertices:

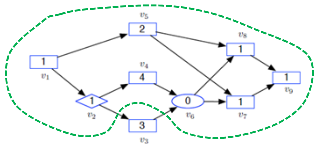
$$vol_i = \sum_{v_{i,j} \in V_i} C_{i,j}$$



It also represents the maximum amount of workload generated by a single instance of a DAG-task

4. Worst-case workload

In the **presence** of conditional branches, the worst-case workload of a task is the worst-case execution time needed to complete it on a dedicated single-core platform, *over all combination of choices for the conditional branches*

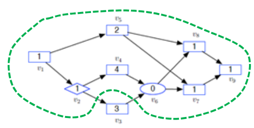


It also represents the maximum amount of workload generated by a single instance of a cp-task

In this example, the worst-case workload is given by all the vertices except v_3, since the branch corresponding to v_4 yields a larger workload

4. Worst-case workload

How can it be computed?



Algorithm 1 Worst-Case Workload Computation

```
procedure WCW(G)
    σ ← TOPOLOGICALORDER(G)
    for z = |V| down to 1 do                 reverse topological order
        i ← σ(z)                             i takes the z^th element of the permutation
        S(v_i) ← {v_i}                       S takes the accumulated worst-case workload from v_i till the end of the cp-DAG
        if SUCC(v_i) ≠ ∅ then                if the vertex has some successors
            if ISBEGINCOND(v_i) then         if the vertex is the head node of a conditional pair
                v* ← argmax_{v∈succ(v_i)} C(S(v))    v* is the successor of v_i achieving the largest partial workload
                S(v_i) ← S(v_i) ∪ S(v*)      S(v*) is merged into S(v_i)
            else                             if instead the vertex is a regular one
                S(v_i) ← S(v_i) ∪ ⋃_{v∈succ(v_i)} S(v)   the workload contribution of all successors is merged into S(v_i)
            end if
        end if
    end for
    return C(S(v_σ(1)))                      the worst-case workload accumulated by the source vertex is returned as output
end procedure
```

4. Worst-case workload

- What is the complexity of this algorithm?

Algorithm 1 Worst-Case Workload Computation

```
procedure WCW(G)
    σ ← TOPOLOGICALORDER(G)
    for z = |V| down to 1 do
        i ← σ(z)
        S(v_i) ← {v_i}
        if SUCC(v_i) ≠ ∅ then
            if ISBEGINCOND(v_i) then
                v* ← argmax_{v∈SUCC(v_i)} C(S(v))
                S(v_i) ← S(v_i) ∪ S(v*)
            else
                S(v_i) ← S(v_i) ∪ ⋃_{v∈SUCC(v_i)} S(v)
            end if
        end if
    end for
    return C(S(v_σ(1)))
end procedure
```

- $O(|E|)$ set operations
- Any of them may require to compute $C(S(v_i))$, which has cost $O(|V|)$

The time complexity is then $O(|E||V|)$

5. Critical chain

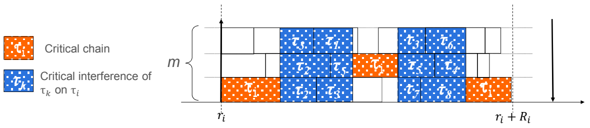
- Given a set of cp-tasks and a (work-conserving) scheduling algorithm, the **critical chain** λ_i^* of a cp-task τ_i is the chain of vertices of τ_i that leads to its worst-case response-time R_i
- How can it be identified?
  - We should know the worst-case instance of τ_i (i.e., the job of τ_i that has the largest response-time in the worst-case scenario)
  - Then we should take its sink vertex v_{i,n_i} and recursively pre-pend the last to complete among the predecessor nodes, until the source vertex $v_{i,1}$ has been included in the chain

Key observation: the critical chain is unknown, but is always upper-bounded by the longest path of the cp-task!

Critical interference

To find the response-time of a cp-task, it is sufficient to characterize the maximum interference suffered by its critical chain

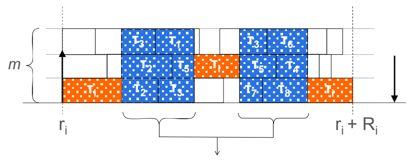
The **critical interference** $I_{i,k}$ imposed by task τ_k on task τ_i is the cumulative workload executed by vertices of τ_k while a node belonging to the critical chain of τ_i is ready to execute but is not executing



Work-conserving schedulers

Global schedulers are typically work-conserving (e.g., Global FP/EDF)

Property: a ready job cannot execute only if **all *m* processors are busy**

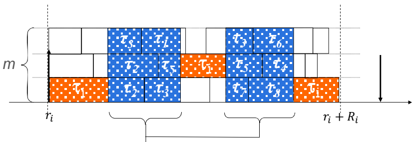


$$I_i = \frac{\sum_{\tau_k} I_{i,k}}{m}$$

We can safely assume that the interference is distributed across all *m* processors

Critical interference

- I_i: total interference suffered by task τ_i
- $I_{i,k}$: total interference of task τ_k on task τ_i



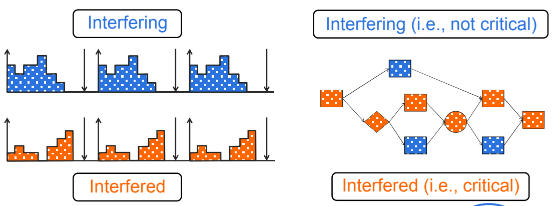
$$I_i = \frac{\sum_{\tau_k} I_{i,k}}{m}$$ For any work-conserving algorithm!

$$R_i = len(\lambda_i^*) + I_i = len(\lambda_i^*) + \frac{\sum_{\tau_k} I_{i,k}}{m}$$

Types of interference

We need to deal with two types of interference:

- **Inter-task interference**: from other tasks in the system; analogous to the classic notion
- **Intra-task interference**: from vertices of the same task on itself; peculiar to parallel tasks only

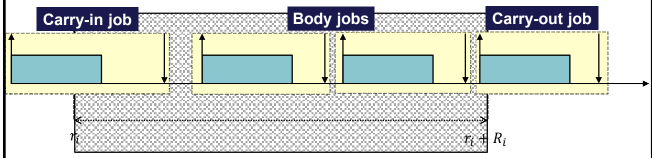


$$R_i = len(\lambda_i^*) + I_i = len(\lambda_i^*) + \frac{\sum_{\tau_k} I_{i,k}}{m} = len(\lambda_i^*) + \frac{1}{m} I_{i,i} + \frac{\sum_{\tau_{k \neq i}} I_{i,k}}{m}$$

Intra-task int. inter-task int.

Inter-task interference

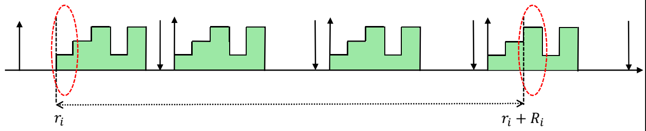
- Caused by other cp-tasks executing in the system
- Finding it exactly is difficult
- We need to find an **upper-bound on the workload** of an interfering task in the scheduling window $[r_i, r_i + R_i]$
- In the sequential case (global multiprocessor scheduling):



What is the scenario that maximizes the interfering workload?

Inter-task interference

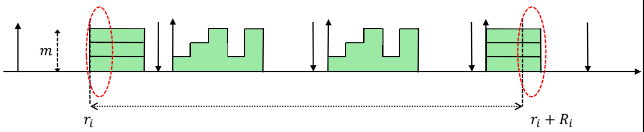
- **Sequential case**
  - The first job of τ_k starts executing as late as possible, with a starting time aligned with the beginning of the scheduling window
  - Later jobs are executed as soon as possible
- **Parallel case**
  - This scenario may not give a safe upper-bound on the interfering workload. Why?



Shifting right the scheduling window may give a larger interfering workload!

Inter-task interference

- **Pessimistic assumption**
  - Each interfering job of task τ_k executes for its worst-case workload W_k
  - The carry-in and carry-out contributions are evenly distributed among all m processors
  - Distributing them on less processors cannot increase the workload within the window
  - Other task configurations cannot lead to a higher workload within the window



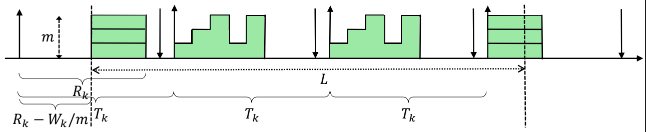
Inter-task interference

- **Lemma:** An upper-bound on the workload of an interfering task τ_k in a scheduling window of length L is given by

$$\mathcal{W}_k(L) = \left\lfloor \frac{L + R_k - W_k/m}{T_k} \right\rfloor W_k + \min\left(W_k, m \cdot \left(\left(L + R_k - \frac{W_k}{m}\right) mod\ T_k\right)\right)$$

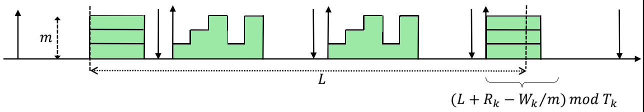
- **Proof:**
  - The maximum number of carry-in and body instances within the window is

$$\left\lfloor \frac{L + R_k - W_k/m}{T_k} \right\rfloor$$



Inter-task interference

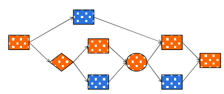
- **Proof (continued):** $\mathcal{W}_k(L) = \left\lfloor \frac{L + R_k - W_k/m}{T_k} \right\rfloor W_k + \min\left(W_k, m \cdot \left(\left(L + R_k - \frac{W_k}{m}\right) mod\ T_k\right)\right)$
  - Each of the $\left\lfloor \frac{L+R_k-W_k/m}{T_k} \right\rfloor$ instances contributes for W_k
  - The portion of the carry-out job included in the window is $\left(L + R_k - \frac{W_k}{m}\right) mod\ T_k$



  - At most m processors may be occupied by the carry-out job
  - The carry-out job cannot execute for more than W_k units

Intra-task interference

It is the interference from vertices of the same task on itself

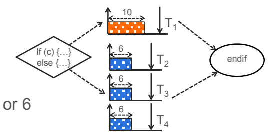


Who is **interfering** and who is **interfered**?

- The **interfered** contribution is the **critical chain**
- **Critical chain**: chain that leads to the WCRT of the cp-task

Critical chain ≠ longest path

- Longest path is 10 time-units
- Critical chain can be either 10 or 6



Intra-task interference

- **Simple upper-bound**

$$R_i = len(\lambda_i^*) + I_i = len(\lambda_i^*) + \frac{1}{m} I_{i,i} + \frac{\sum_{\tau_{k \neq i}} I_{i,k}}{m}$$

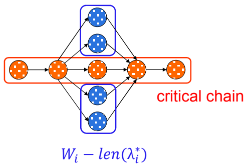
$$I_{i,k}(L) \leq \mathcal{W}_k(L)$$

$$Z_i \stackrel{\text{def}}{=} len(\lambda_i^*) + \frac{1}{m} I_{i,i}$$

$$\leq len(\lambda_i^*) + \frac{1}{m}(W_i - len(\lambda_i^*))$$

$$\leq L_i + \frac{1}{m}(W_i - L_i)$$

Length of the longest path



Putting things together

- **Schedulability condition**

Given a cp-task set globally scheduled on m processors, an upper-bound R_i^{ub} on the response-time of a task τ_i can be derived by the fixed-point iteration of the following expression, starting with $R_i^{ub} = L_i$:

$$R_i^{ub} = L_i + \frac{1}{m}(W_i - L_i) + \frac{1}{m} \sum_{\forall k \neq i} \mathcal{X}_k^{ALG}$$

- **Global FP**

$$\mathcal{X}_k^{ALG} = \mathcal{X}_k^{FP} = \begin{cases} \mathcal{W}_k(R_i^{ub}), & \forall\, k < i \\ 0, & otherwise \end{cases}$$

Decreasing priority order

- **Global EDF**

$$\mathcal{X}_k^{ALG} = \mathcal{X}_k^{EDF} = \mathcal{W}_k(R_i^{ub}), \forall\, k \neq i$$

$$\mathcal{W}_k(L) = \left\lfloor \frac{L + R_k - W_k/m}{T_k} \right\rfloor W_k + \min\left(W_k, m \cdot \left(\left(L + R_k - \frac{W_k}{m}\right) \bmod T_k\right)\right)$$

Putting things together

$$R_i^{ub} = L_i + \frac{1}{m}(W_i - L_i) + \frac{1}{m} \sum_{\forall k \neq i} \mathcal{X}_k^{ALG}$$

- **Global FP**

The fixed-point iteration updates the bounds in decreasing priority order, starting from the highest priority task, until either:

- one of the response-time bounds exceeds the task relative deadline D_i (negative schedulability result);
- OR no more update is possible (positive schedulability result), i.e., $\forall\, i$: $R_i^x = R_i^{x+1} \leq D_i$

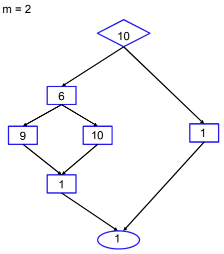
- **Global EDF**
  - Multiple rounds may be needed

Reference

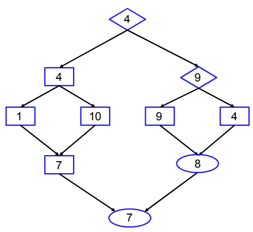
A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, G. Buttazzo, ***Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems***, Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)

Schedulability example

Global FP
m = 2



High-priority task
D = 35
T = 37
Len = 28
W = 37



Low-priority task
D = 139
T = 229
Len = 37
W = 37

Solution sketch

$R_1^{(1)} = L_1 = 28$

$R_1^{(2)} = L_1 + \frac{1}{m}(W_1 - L_1) = 32.5$

$R_1^{(3)} = R_1^{(2)} = 32.5 \leq 35$ Task 1 is schedulable

$R_2^{(1)} = L_2 = 37$

$R_2^{(2)} = L_2 + \frac{1}{m}(W_2 - L_2) + \frac{1}{m}\left(\left\lfloor \frac{R_2 + R_1 - W_1/m}{T_1} \right\rfloor W_1 + \min(\ldots)\right) = 69.5$

$R_2^{(3)} = 92.5$

$R_2^{(4)} = R_2^{(3)} = 92.5 \leq 139$ Task 2 is schedulable

Thank you!

Alessandra Melani
alessandra.melani@sssup.it

Retis
Real-Time Systems Laboratory

49