IN4343 Real-Time Systems, Lecture 8

Simulation-Based Tests and Exact Schedulability Analysis

Contact

m.nasri@tudelft.nl

(appointment by email)
Sources

There is always FUN for those who seek it in their life!

Paper 1:

Paper 2:

Paper 3 (optional) on multiprocessor scheduling:

Giorgio Buttazzo’s book: chapter 8
Agenda

• **Successful stories on non-preemptive scheduling**
  • Precautious-RM
  • Critical-Window EDF (CW-EDF)

• Quiz with bonus points

• Exact schedulability analysis *(for non-preemptive scheduling)*

• Limited-preemptive scheduling
A necessary test for non-preemptive scheduling

Is this task set feasible?

No! Whatever you do, it is impossible to find a feasible schedule for this task set.

Let’s design a necessary test together

\[ U \leq 1 \]
\[ \forall i, 1 < i \leq n, \]

\[ C_i \leq 2(T_1 - C_1) \]

Can we improve it?

Maybe! No body knows!

Assuming that tasks are sorted by their period
Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:

Why there is a deadline miss?
Because at time 4, NP-FP took a wrong decision!
This wrong decision resulted in a loooong blocking time on $J_{1,2}$
Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:

\[ C_1 = 1 \]
\[ C_2 = 3 \]
\[ C_3 = 8 \]

 Which task do you think will likely have a lot of deadline misses in the future?

\( \tau_1 \) because it has the smallest period (highest frequency of activation)
Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:

How could this situation be avoided? By not scheduling $J_{3,1}$ at time 4
Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:

How should the scheduler know that it MUST leave the processor “idle” instead of executing a task? (try to make a general rule)

One of the answers to this question became a journal paper! Each of your answers can end up the same way if you want!
By looking to the future!

How?

By fortune telling! Obviously :D

Using the “knowledge” about the periodicity of the tasks!
Why non-preemptive scheduling is hard?

NP-FP (rate-monotonic priorities) schedule:

Knowing tasks are periodic, we exactly know when the "future jobs" of the tasks will arrive.

Hence, we can check if our current decision will impact the deadline of those FUTURE jobs!

Example: check whether or not, executing the current highest-priority ready task will cause a deadline miss for the next job of $\tau_1$
A general non-work-conserving solution

1- Select the highest-priority ready job
   (the highest-priority job can be chosen by any scheduling policy of your choice: EDF, FP, etc.)

2- Check if executing that job will cause a deadline miss for a set of jobs that will arrive in the future
   • Yes? Then don’t schedule the current task, instead, leave the processor idle until the next job arrives to the system
   • No? Then schedule the current high-priority ready task

Why just a “set of future jobs”? Why not for “all future jobs”?
Example: Precautious-RM policy
(just look at “one” future job)

- **Select** the highest-priority ready task (using RM priorities)
- **If** (it will cause a deadline miss for the next job of \( \tau_1 \)) **then**
  leave the processor idle
- **Else**
  Dispatch the task

![Diagram]

- \( \tau_1 \)
- \( \tau_2 \)
- \( \tau_3 \)

\[ t = 4 \]

\[ C_1 = 1 \]
\[ C_2 = 3 \]
\[ C_3 = 8 \]
Example: Precautious-RM policy
(just look at “one” future job)

- **Select** the highest-priority ready task (using RM priorities)
- **If** (it will cause a deadline miss for the next job of $\tau_1$) **then**
  leave the processor idle
- **Else**
  Dispatch the task

\[ C_1 = 1 \]
\[ C_2 = 3 \]
\[ C_3 = 8 \]
Example: Precautious-RM policy
(just look at “one” future job)

- Select the highest-priority ready task (using RM priorities)
- If (it will cause a deadline miss for the next job of \( \tau_1 \)) then leave the processor idle
- Else
  Dispatch the task

\( \tau_1 \)
\( \tau_2 \)
\( \tau_3 \)

\( t = 5 \)

\( C_1 = 1 \)
\( C_2 = 3 \)
\( C_3 = 8 \)
Example: Precautious-RM policy (just look at “one” future job)

- **Select** the highest-priority ready task (using RM priorities)
- If (it will cause a **deadline miss** for the next job of $\tau_1$) then leave the processor idle
- Else
  Dispatch the task

### Open problem: how can we improve schedulability while keeping the solution $O(1)$?

### Open problem: how can we design a non-work-conserving algorithm for a multiprocessor system?
Critical-Window EDF (CW-EDF)

- Considers the next job of each task
- Sorts their deadlines
- Obtains the latest start time of each job starting from the one with the latest deadline
- Checks if the current high priority job can finish before the latest start time $S$

$\tau_6$ ready
$\tau_5$ ready
$\tau_4$ completed
$\tau_3$ completed
$\tau_2$ completed
$\tau_1$ completed

done_{\text{next}}^i(t) = \text{the absolute deadline of the next job of Task } i \text{ which will be released after } t.

[ECRTS 2016]
Important design factors for scheduling algorithms

• Efficiency
  • How successful is the algorithm in generating feasible schedules

• Analyzability
  • Can you analyze it at all? (can you come up with a schedulability test?)

• Runtime

• Memory requirement
0.1 bonus points for the 3 winners and for anyone who answers all questions correctly

(how should we figure out who is who?) :D

Step 1: open www.kahoot.it in your browser (phone or laptop)
Step 2: enter the pin code and then a nickname
Agenda

• Successful stories on non-preemptive scheduling
  • Precautious-RM
  • Critical-Window EDF (CW-EDF)

• **Exact schedulability analysis** (for non-preemptive scheduling)

• Limited-preemptive scheduling
How to design a generalize yet exact schedulability test?

for a wide class of scheduling policies

A test that is both necessary and sufficient
A closer look at the “scheduling”

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Exact execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₃</td>
<td>30</td>
<td>13</td>
</tr>
<tr>
<td>τ₂</td>
<td>30</td>
<td>8</td>
</tr>
<tr>
<td>τ₁</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Consider non-preemptive fixed-priority scheduling (NP-FP)

Both existing tests for NP-FP and NP-EDF that you learned in Lecture 7 reject this task set.
The **dumbest yet smartest** schedulability test
(for periodic tasks)

- **Schedule** the tasks until the **hyperperiod** with the given scheduling policy
- If there is a **deadline miss**, it is **not schedulable**!
- Otherwise, it is **schedulable**!

**Why is it smart (and helpful)?**

Because it allows designing schedulability analyses for a wide class of scheduling policies and task sets
The **dumbest yet smartest** schedulability test
(for periodic tasks)

- **Schedule** the tasks until the **hyperperiod** with the given scheduling policy
- If there is a **deadline miss**, it is **not schedulable**!
- Otherwise, it is **schedulable**!

**Why is it dumb?**

Because some task parameters are **non-deterministic**!
For example, we do not know what is the **exact execution time** or **release time** of the tasks at runtime!
The **dumbest yet smartest** schedulability test
(for periodic tasks)

- **Schedule** the tasks until the **hyperperiod** with the given scheduling policy
- If there is an **deadline miss**, it is **not schedulable**!
- Otherwise, it is **schedulable**!

This is called **simulation-based** schedulability test
Example: the effect of non-deterministic execution time

NP-FP (rate-monotonic priorities) schedule:

Offline scheduling (using the WCET of the tasks) tells us that there is no deadline miss!

Is this task set schedulable?
Example: the effect of non-deterministic execution time

NP-FP (rate-monotonic priorities) schedule:

Because there exist this (and many more) scenarios that lead to deadline misses!

Is this task set schedulable?  No!
Key challenges
(with non-deterministic parameters)

<table>
<thead>
<tr>
<th>Job</th>
<th>Release time</th>
<th>Deadline</th>
<th>Execution time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>$J_1$</td>
<td>0</td>
<td>7</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>$J_2$</td>
<td>2</td>
<td>5</td>
<td>15</td>
<td>2</td>
</tr>
</tbody>
</table>

For example, the interrupt service routine (e.g., timer interrupt handler) has a non-deterministic execution time.

Naively enumerating all possible combinations of release times and execution times (a.k.a. execution scenarios) is not practical.

Arrival time is the “expected” time at which the task must arrive to the system.
Release time is the “actual time” at which the task enters the system.
### An example with execution-time variation and release jitter

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Release Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_3$</td>
<td>30</td>
<td>[3, 13]</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>30</td>
<td>[7, 8]</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>10</td>
<td>[1, 2]</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**Table: Execution Time**

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Release Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_3$</td>
<td>30</td>
<td>[3, 13]</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>30</td>
<td>[7, 8]</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>10</td>
<td>[1, 2]</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**Graph 1:**
- $\tau_3$: 13
- $\tau_2$: 8
- $\tau_1$: 2

**Graph 2:**
- $\tau_3$: 13
- $\tau_2$: 7
- $\tau_1$: Missed

**Assumption:**
- $a_{3,1} = 13$
- $\forall k, C_{i,k} = C_i$
  (every task executes for $C_i$ units)

**Deadline Miss:**
- $a_{3,1} = 5$
- $C_{2,1} = 7$
An example with execution-time variation and release jitter

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Release jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_3$</td>
<td>30</td>
<td>[3, 13]</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>30</td>
<td>[7, 8]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>10</td>
<td>[1, 2]</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Assumption:

$a_{3,1} = 13$

$\forall k, C_{i,k} = C_i$

(every task executes for $C_i$ units)

More than 100 different schedules

Not schedulable

Checking all possible schedules

Now let’s try to do the dumb job smartly!
An example with execution-time variation and release jitter

### Execution time

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Release jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_3$</td>
<td>30</td>
<td>$[3, 13]$</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>30</td>
<td>$[7, 8]$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>10</td>
<td>$[1, 2]$</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

#### Assumption:

\[ a_{3,1} = 13 \]
\[ \forall k, C_{i,k} = C_i \]

(every task executes for $C_i$ units)

#### Deadline miss:

\[ a_{3,1} = 5 \]
\[ C_{2,1} = 7 \]

#### Try to draw the graph of job orderings

1. The finish time of $J_{1,1}$ and $J_{2,1}$ is **not** smaller than 10
2. The finish time of $J_{1,1}$ and $J_{2,1}$ is **smaller** than 10

#### More than 100 different schedules

#### Not schedulable

#### Only two different job orderings
Challenges

An exact analysis must consider all possible execution scenarios (i.e., combination of release times and execution times).

Due to scheduling anomalies

Observation

There are fewer permissible job orderings than schedules

Research question

Is there a way to use job-ordering abstraction to analyze schedulability?

Research questions

How to abstract schedules in a graph of job orderings?
How to efficiently find all job orderings?
How to identify timing violations in the resulting graph?
Abstracting schedules in a graph of job orderings

**Requirement**
Knowing if there is a deadline miss

**Solution**
Encode the **earliest** and **latest finish time** of each job

**Verification of schedulability**
Check if the **latest finish time** is not larger than the deadline

Each path shows a job ordering

**NOTE:** this is 11 = 8 + 3 (not 12)

### Execution time

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_3$</td>
<td>30</td>
<td>[3, 13]</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>30</td>
<td>[7, 8]</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>10</td>
<td>[1, 2]</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Schedule-Abstraction Graph
(definition, usage, and construction)
What is a schedule-abstraction graph?

“schedule abstraction” is a technique that allows aggregating “similar” schedules while searching for all possible schedules.

What is a schedule-abstraction graph?

A path in the graph represents an ordered set of dispatched jobs.

Initial state: no job has been dispatched.

Final state: Every path includes all jobs.

What is a schedule-abstraction graph?

A path in the graph represents an ordered set of dispatched jobs.
A vertex abstracts a system state.
An edge abstracts a dispatched job.

system state (before dispatching $J_3$)

system state (after dispatching $J_3$)

$J_3$ finishes any time during [5, 10]

What is a schedule-abstraction graph?

A **path** in the graph represents an ordered set of dispatched jobs.

A **vertex** abstracts a system state. An **edge** abstracts a dispatched job.

A **state** represents the finish-time interval of any path reaching that state.

![Diagram of a schedule-abstraction graph]

$J_1$ finishes in $[10, 15]$

$J_2$ finishes in $[12, 25]$

$\nu_p$: $[10, 25]$

What is a schedule-abstraction graph?

- A path in the graph represents an ordered set of dispatched jobs.
- A vertex abstracts a system state.
- An edge abstracts a dispatched job.
- A state represents the finish-time interval of any path reaching that state.

A vertex $v_p$: [10, 25]

- $J_1$ finishes in [10, 15], so the processor is certainly busy before time 10.
- $J_2$ finishes in [12, 25], so the processor is certainly available after time 25.

How to use a schedule-abstraction graph?

The **worst-case (best-case) response time** of a job $J_i$ is its largest (smallest) finish time among all edges whose label is $J_i$.

**Example for job $J_2$**

$J_2$ finishes in [12, 25]

$J_2$ finishes in [3, 10]

$J_2$ finishes in [20, 24]

$J_2$ finishes in [14, 18]

$J_2$ finishes in [16, 28]

BCRT = 3

WCRT = 28

How to build a schedule-abstraction graphs?

[RTSS’17] used a breadth-first strategy

Repeat until every path includes all jobs
1. Find the shortest path
2. For each not-dispatched job that can be dispatched after the path:
   2.1. **Expand** (add a new vertex)
   2.2. **Merge** (if possible, merge the new vertex with an existing vertex)

Expanding the graph

Is there a path that can be expanded?

Yes

Select the shortest path $P$

Find eligible jobs

For each eligible job, find the earliest and latest finish time and add them to $P$

Merge any two paths that share the same set of jobs

Report deadline misses

No

An eligible job for path $P$ is a job that can be scheduled after $P$ in at least one execution scenario

Path $P$

$v_1$ ... $v_i$ $J_3$ $v_i$ $[e_i, l_i]$ $J_4$ $v_j$

Priority

High

Low

$e_i = \text{the earliest finish time of path } P$

$l_i = \text{the latest finish time of path } P$
Expanding the graph

An eligible job for path $P$ is a job that can be scheduled after $P$ in at least one execution scenario.

For each eligible job, find the earliest and latest finish time and add them to $P$.

Merge any two paths that share the same set of jobs.

Report deadline misses.

$e_i = \text{the earliest finish time of path } P$

$l_i = \text{the latest finish time of path } P$
Expanding the graph

An **eligible job** for path \( P \) is a job that can be scheduled after \( P \) in **at least one execution scenario**

---

**Graph Expansion Algorithm**

1. **Is there a path that can be expanded?**
   - Yes: **Select the shortest path** \( P \)
   - No: **Find eligible jobs**

2. **Find eligible jobs**
   - For each eligible job, find the earliest and latest finish time and add them to \( P \)

3. **Merge** any two paths that share the same set of jobs

4. **Report deadline misses**

---

**Mathematical Formulation**

- \( \vec{e}_i = \text{the earliest finish time of path } P \)
- \( \vec{l}_i = \text{the latest finish time of path } P \)

---

**Graph Example**

- **Path** \( P \)
- **Eligible job** \( J_2 \)
- **Time**
  - \( t_1 \)
  - \( t_2 \)
  - \( t_4 \)
An example
Recall: schedule abstraction graph

Which job will be the first to execute?

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_3)</td>
<td>30</td>
<td>[3, 13]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>30</td>
<td>[7, 8]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(\tau_1)</td>
<td>10</td>
<td>[1, 2]</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Recall: schedule abstraction graph

Consider non-preemptive fixed-priority (NP-FP) algorithm
Recall: schedule abstraction graph

Consider non-preemptive fixed-priority (NP-FP) algorithm

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_3)</td>
<td>30</td>
<td>[3, 13]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>30</td>
<td>[7, 8]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(\tau_1)</td>
<td>10</td>
<td>[1, 2]</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Recall: schedule abstraction graph

Consider non-preemptive fixed-priority (NP-FP) algorithm

Why the latest start time of $J_{3,1}$ in scenario 2 is **not** 10?

Because then it becomes Scenario 1: at time 10, $J_{3,1}$ is not the highest-priority task

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_3$</td>
<td>30</td>
<td>[3, 13]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>30</td>
<td>[7, 8]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>10</td>
<td>[1, 2]</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Recall: schedule abstraction graph

Why the latest start time of $J_{3,1}$ in scenario 2 is not 10?

Because then it becomes Scenario 1: at time 10, $J_{3,1}$ is not the highest-priority task.
Recall: schedule abstraction graph

Consider non-preemptive fixed-priority (NP-FP) algorithm

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_3)</td>
<td>30</td>
<td>3</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>30</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>(\tau_1)</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Execution time

\(\tau_3\):

\(\tau_2\):

\(\tau_1\):
Recall: schedule abstraction graph

Consider non-preemptive fixed-priority (NP-FP) algorithm

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₃</td>
<td>30</td>
<td>[3, 13]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>τ₂</td>
<td>30</td>
<td>[7, 8]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>τ₁</td>
<td>10</td>
<td>[1, 2]</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Rules to generate a schedule-abstraction graph
Deriving rules for expanding the graph

The processor can become available at any time in this interval

Q1: Can $J_{low}$ be the first job that is being scheduled next?

Q2: if yes, then what is the earliest start (and finish) time and latest start (and finish) time of $J_{low}$ such that it is scheduled before $J_{high}$?

<table>
<thead>
<tr>
<th>Job</th>
<th>Release time</th>
<th>Deadline</th>
<th>Execution time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>$J_{low}$</td>
<td>5</td>
<td>15</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>$J_{high}$</td>
<td>12</td>
<td>20</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
Deriving rules for expanding the graph

Find the earliest start time (EST) of $J_{low}$

Try to imagine the best scenario that can happen for the start time of $J_{low}$

$E_{low}$ is the earliest time at which the job is possibly released and the processor is possibly available

$\text{EST} = \max \{ 5, 10 \} = 10$

<table>
<thead>
<tr>
<th>Job</th>
<th>Release time</th>
<th>Deadline</th>
<th>Execution time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td></td>
<td>Min</td>
</tr>
<tr>
<td>$J_{low}$</td>
<td>5</td>
<td>15</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>$J_{high}$</td>
<td>12</td>
<td>20</td>
<td>45</td>
<td>1</td>
</tr>
</tbody>
</table>
Deriving rules for expanding the graph

\( J_{\text{low}} \)

Find the **earliest start time** (EST) of \( J_{\text{low}} \)

\( J_{\text{high}} \)

Find the **latest start time** (LST) of \( J_{\text{low}} \) for a work-conserving and a job-level fixed-priority scheduling policy

1. **Work-conserving rule**

\( t_{\text{wc}} \) is the time at which a job is certainly released and the processor is certainly available

\[ t_{\text{wc}} = \max\{15, 25\} = 25 \]

Try to imagine a scenario in which there will be a certainly released job and a certainly available processor.
Deriving rules for expanding the graph

$v_p$: 

$J_{low}$ 

$J_{high}$ 

1. Work-conserving rule
$t_{wc}$ is the time at which a job is certainly released and the processor is certainly available
$t_{wc} = \max \{15, 25\} = 25$

2. JLFP rule
$t_{high}$ is the time at which a higher-priority job is certainly released
$t_{high} = 20$

<table>
<thead>
<tr>
<th>Job</th>
<th>Release time</th>
<th>Deadline</th>
<th>Execution time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>$J_{low}$</td>
<td>5</td>
<td>15</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>$J_{high}$</td>
<td>12</td>
<td>20</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
Deriving rules for expanding the graph

Find the earliest start time (EST) of $J_{low}$

Find the latest start time (LST) of $J_{low}$ for a work-conserving and a job-level fixed-priority (JLFP) scheduling policy

1. Work-conserving rule

$t_{wc}$ is the time at which a job is certainly released and the processor is certainly available

$t_{wc} = \max \{15, 25\} = 25$

2. JLFP rule

$t_{high}$ is the time at which a higher-priority job is certainly released

$t_{high} = 20$

3. Latest start time (LST)

$LST = \min\{t_{high} - 1, t_{wc}\}$

$= \min\{20 - 1, 25\} = 19$
Deriving rules for expanding the graph

\[ \nu_p: [10, 25] \quad J_{low} \quad \nu_q: [12, 24] \]

1. Find the earliest start time (EST) of \( J_{low} \)
2. Find the latest start time (LST) of \( J_{low} \) for a work-conserving and a job-level fixed-priority (JLFP) scheduling policy
3. If \( \text{EST} \leq \text{LST} \) then add an edge for job \( J_{low} \)

\[ \text{Earliest finish time (EFT)} = \text{EST} + \text{BCET} \]
\[ \text{EFT} = 10 + 2 = 12 \]

\[ \text{Latest finish time (LFT)} = \text{LST} + \text{WCET} \]
\[ \text{LFT} = 19 + 15 = 24 \]
More details?

In the paper
What did we learn from this technique?

How to **automate** the exploration of all possible schedules!

Doing the dumb job smartly!