Mitra Nasri
m.nasri@tue.nl
Assistant professor
Electronic Systems Group
Department of Electrical Engineering

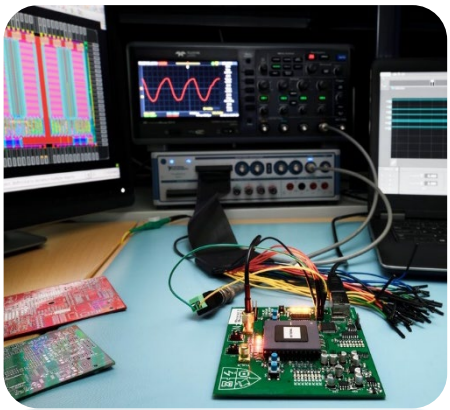
EINDHOVEN UNIVERSITY OF TECHNOLOGY

Schedule-Abstraction Graph:

An Efficient Technique for Response-Time Analysis in Real-Time Systems

Department of Electrical Engineering

Electronic Systems (ES) Group



Real-time systems

Systems that require both **functional correctness** and **temporal correctness**

Real-time systems

Systems that require both **functional correctness** and **temporal correctness**



Safety

- Human life
- Environment

Time-predictability

- A late (or missed) actuation may cause safety violation
- Example: breaking, air-bag inflation, etc.

Correct response

Functional correctness

+

Temporal correctness

On time response

Fast ≠ predictable

Real-time systems

Systems that require both **functional correctness** and **temporal correctness**

Real-time does not mean "fast"
or having deadlines in milliseconds!

It is about timing predictability
(Deadlines might be in the order of seconds or minutes)

Safety

- Human life
- Environment

Time-predictability

- A late (or missed) actuation may cause safety violation
- Example: breaking, air-bag inflation, etc.

Correct response

Functional correctness

+

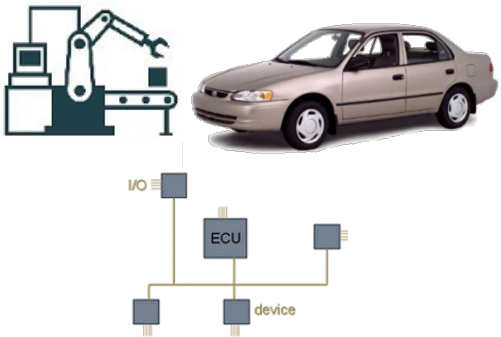
Temporal correctness

On time response

Fast ≠ predictable

Why guaranteeing temporal correctness is hard?

Back then

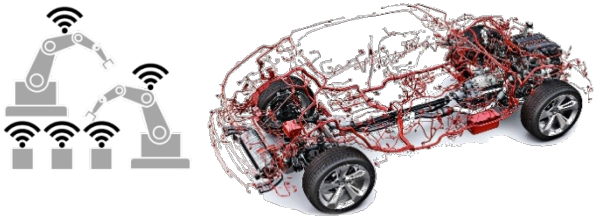


A few computing nodes and control loops

Simple hardware and software **architecture**

Simple, predictable, and easier-to-analyze

Now (and future)



Complex software (application workloads) running on heterogeneous computing environment

Intensive I/O accesses

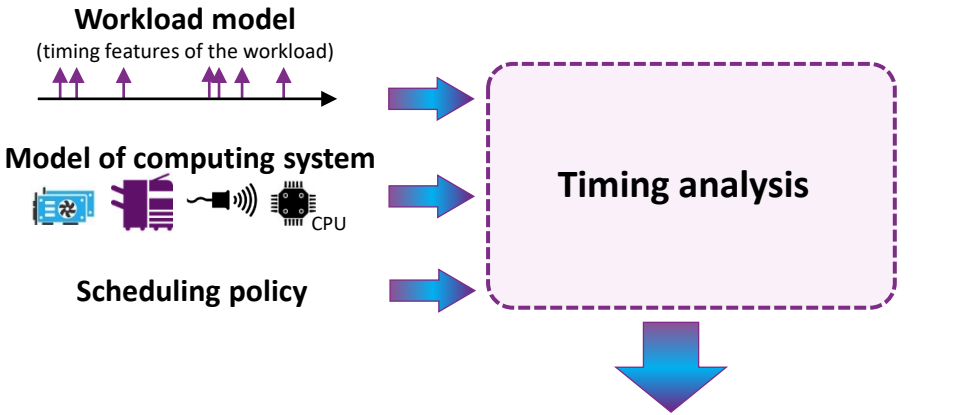
Use of hardware accelerators (GPUs, FPGA, co-processors, etc.)

Computation offloading (to the cloud, edge, etc.)

Heterogeneous communication

Complex, less predictable, and harder-to-analyze

Timing analysis



Response-time analysis: where are we?

Closed-form analyses
(e.g., problem-window analysis)

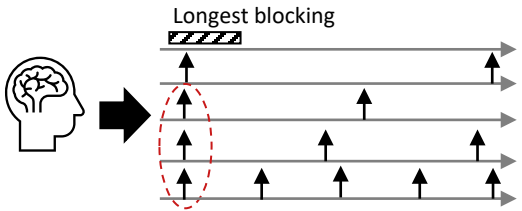
- Fast
- Pessimistic
- Hard to extend

$$R_i^{(0)} = C_i + \sum_{j=1}^{i-1} C_j$$

$$R_i^{(k)} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil C_j$$

...lority tasks. A response-time ... set with a limited-preemptive ... is computed by iterating the ... point is reached, starting with ... $len(G_k)$):

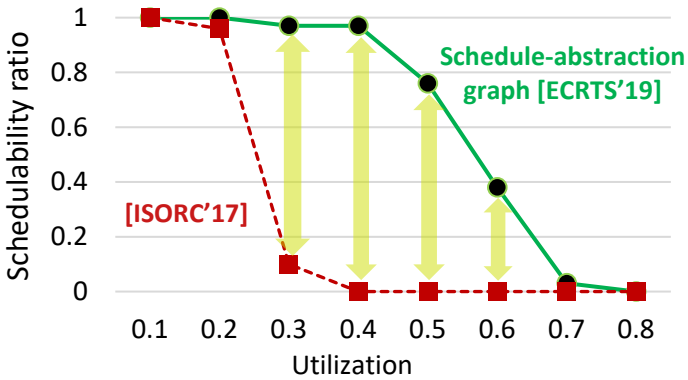
$$R_k \leftarrow len(G_k) + \frac{1}{m}(vol(G_k) - len(G_k) + I_k^{hp} + I_k^{lp}) \quad (1)$$



Where has it taken us?

Experiment: limited-preemptive scheduling of parallel DAG tasks

Setup: 8 cores, 10 parallel DAG tasks



[ISORC'17]: Serrano et al., "An Analysis of Lazy and Eager Limited Preemption Approaches under DAG Based Global Fixed Priority Scheduling", ISORC, 2017.

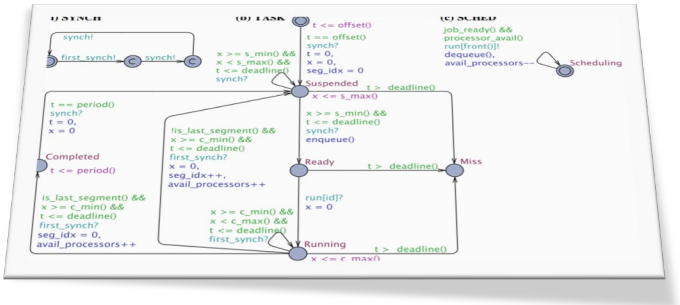
Response-time analysis: where are we?

Closed-form analyses
(e.g., problem-window analysis)

- Fast
- Pessimistic
- Hard to extend

Exact analyses in generic formal verification tools (e.g., UPPAAL)

- Accurate
- Easy to extend
- Not scalable



Schedule-abstraction graph

- Applicable to complex problems
- Easy to extend
- Highly accurate
- Relatively fast

Industrial use cases are typically large, complex, and require accurate analysis

Open source:
https://github.com/gnelissen/np-schedulability-analysis
Publications:
[RTSS'17, ECRTS'18, ECRTS'19, DATE'19, RTSS'20, RTSS'21, RTAS'22, RTNS'22, ECRTS'22, …]

My research in a nutshell

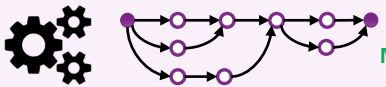
3000x faster than
generic timing verification tools (e.g., UPPAAL)

Workload model
(timing features of the workload)

Resource model

Scheduling algorithm
(dynamic and static scheduling algorithms)

Timing-analysis framework
(it is a formal verification engine dedicated to **timing models** and **timing properties**)



Partial-order reduction

In our RTAS'22 work, we made it
5 orders-of-magnitude faster
using partial-order-reduction

Response-time bounds

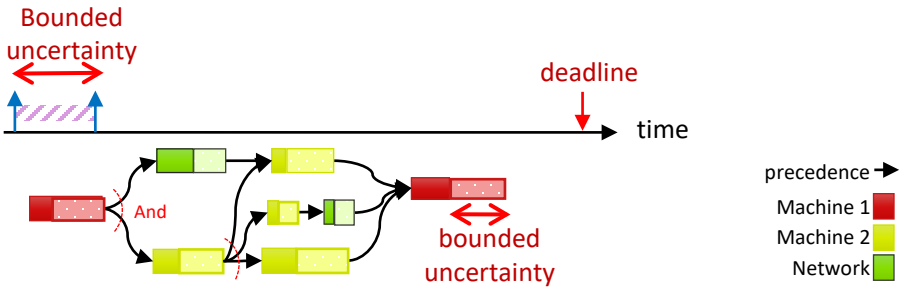
- **Lower and upper bounds on**
  - response time, end-to-end latency, jitter, quality of service/control
- **Counter examples** in case of timing violation

Many top-rank conference papers
[RTSS'17, ECRTS'18, ECRTS'19, DATE'19, RTSS'20, RTSS'21, RTAS'22, RTNS'22, ECRTS'22]

The response-time analysis problem

Workload model

Job
- Arrival model
- Execution model



Resource model

How many resources of which type (and with what access costs) are available?

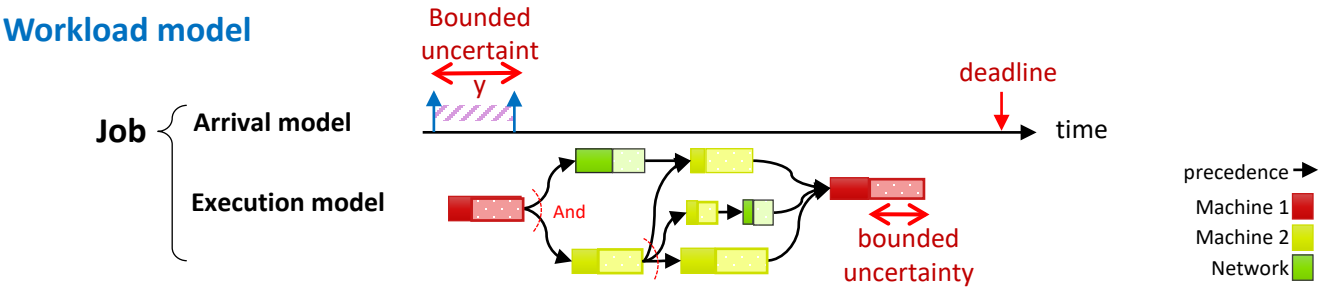
Scheduler model

How are the resources governed (scheduled)?

Response-time analysis problem

Given **a set of jobs, resources, and scheduling policies,**
Determine **the worst-case response time of each job**

Why deriving the worst-case response time is hard when there is uncertainty in timing parameters?



Preliminaries

Scheduling policy

Online scheduling

Schedulability and feasibility of a schedule

Scheduling policy (algorithm)

A **scheduling policy** is an algorithm that determines when (in which intervals) a task can execute on a processor (or resource).

Examples

- **Fixed-task priority scheduling**
  - supported by almost all real-time or embedded operating systems
  - example: Linux, FreeRTOS, RTEMS, …
- **Earliest-deadline first (EDF)**
  - It is known to be an optimal algorithm for preemptive tasks deployed on a single-core platform.
  - Implemented in Linux and many real-time operating systems
- **Round robin**
- **First-in-first-out**
- **Shortest-job first**

Properties of scheduling policies

- **Preemptive** v.s. **non-preemptive**
  - Do you allow preemption? Does the scheduling policy force a task to leave the processor?
- **Work-conserving** v.s. **non-work-conserving**
  - Do you allow the processor to remain idle even if you have some tasks that are pending?
- **Static** v.s. **dynamic priorities**
  - Do the task priorities change at runtime?
- **Offline** v.s. **online**
  - When do you take your decision for executing a task? Before you start the system, or at runtime?
- **Optimal** v.s. **non-optimal** (e.g., heuristic)
  - Does the policy result in an optimal schedule? (e.g., does it guarantee that your system meets its deadlines?)

Static v.s. dynamic priorities

Task-level static

Scheduling decisions are taken based on task's fixed parameters that are known beforehand.

Example: fixed-priority scheduling

Job-level static

Scheduling decisions are taken based on parameters of the job at its release time.

Example: earliest-deadline first (EDF) scheduling uses the absolute deadline of the jobs in order to decide which job has the highest-priority.

> Note: EDF is a task-level dynamic scheduling policy because the priority of a task is determined by the absolute deadline of the current pending job of the task.

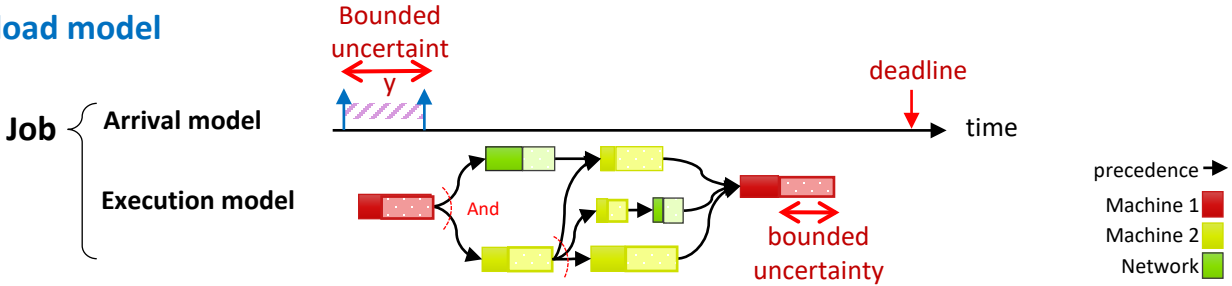
Job-level dynamic

Scheduling decisions are taken based on parameters that can change with time.

Example: shortest-remaining execution-time first policy

Recall: The response-time analysis problem

Workload model



Resource model

How many resources of which type (and with what access costs) are available?

Scheduler model

How are the resources governed (scheduled)?

Response-time analysis problem

Given **a set of jobs, resources, and scheduling policies,**
Determine **the worst-case response time of each job**

Why the problem is hard?

The preemptive version of this problem (which is believed to be easier than the non-preemptive one), is already NP-Complete

One of the simplest forms of the problem:

Response-time analysis problem

Given

- **a set of non-preemptible jobs** (with a given arrival interval, execution time, and deadline)
- **scheduled by a fixed-priority scheduling policy**
- **on a uniprocessor platform,**

Determine

the worst-case response time of each job

Why the problem is hard?

One of the simplest forms of the problem:

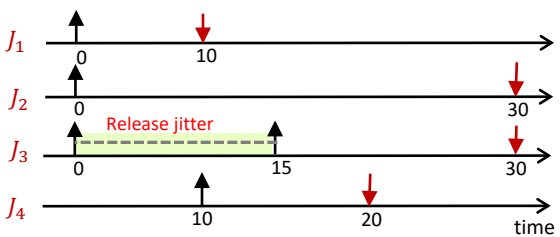
Response-time analysis problem

Given

- **a set of non-preemptible jobs** (with a given arrival interval, execution time, and deadline)
- **scheduled by a fixed-priority scheduling policy**
- **on a uniprocessor platform,**

Determine

the worst-case response time of each job



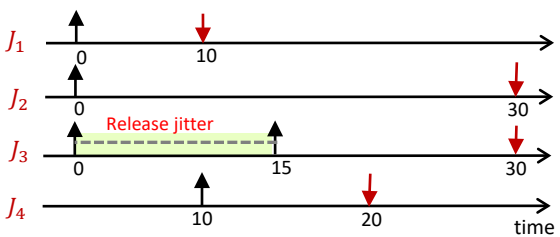
Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_1	0	0	10	1	2	high
J_2	0	0	30	7	8	medium
J_3	0	15	30	3	13	low
J_4	10	10	20	1	2	high

Earliest release time | Latest release time | BCET | WCET

Why the problem is hard?

Goal: find the worst-case response time of each job (for any imaginable schedule that a

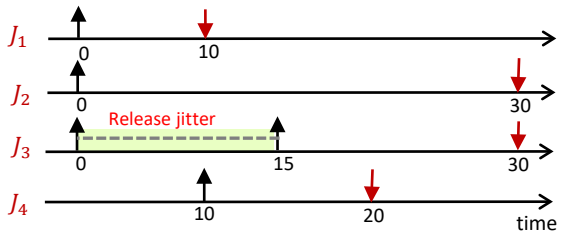
Q: Why can't we "simulate" one schedule using a discrete-even simulator and see if there will be a deadline miss?



Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_1	0	0	10	1	2	high
J_2	0	0	30	7	8	medium
J_3	0	15	30	3	13	low
J_4	10	10	20	1	2	high

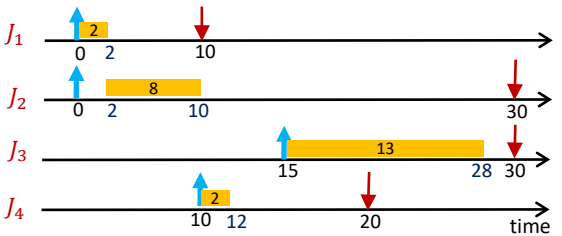
Earliest release time | Latest release time | BCET | WCET

Why the problem is hard?



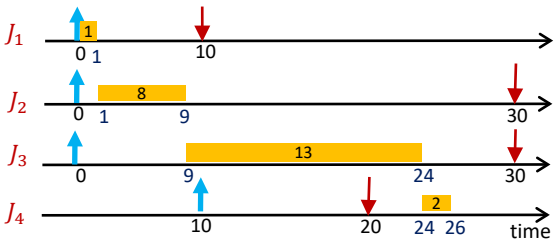
Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_1	0	0	10	1	2	high
J_2	0	0	30	7	8	medium
J_3	0	15	30	3	13	low
J_4	10	10	20	1	2	high

Execution scenario 1: jobs are released very late and have their largest execution time.



No miss

Execution scenario 2: jobs are released very early and have their largest execution time except for J_1.



Deadline miss for J_4

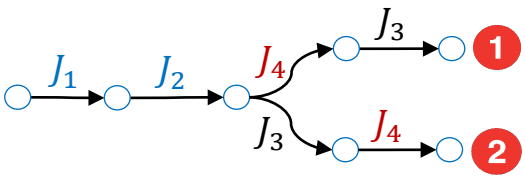
How should we find such a worst-case scenario?

Why the problem is hard?

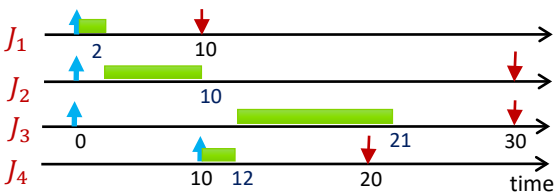
Naively enumerating all possible combinations of release times and execution times (a.k.a. execution scenarios) **is not practical**

Observation:

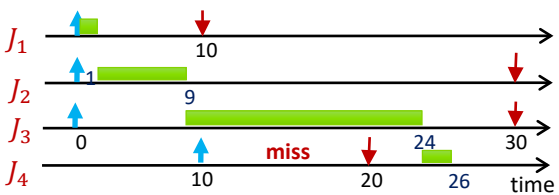
There are fewer permissible job orderings than schedules



Example for path 1



Example for path 2



- **2 possible job ordering**
- **1200 different combinations for release times and execution times**

Why the problem is hard?

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

Observation:

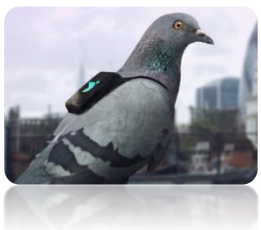
There are fewer permissible job orderings than schedules

Solution idea:

We use job-ordering abstraction to build a graph that abstracts all possible schedules

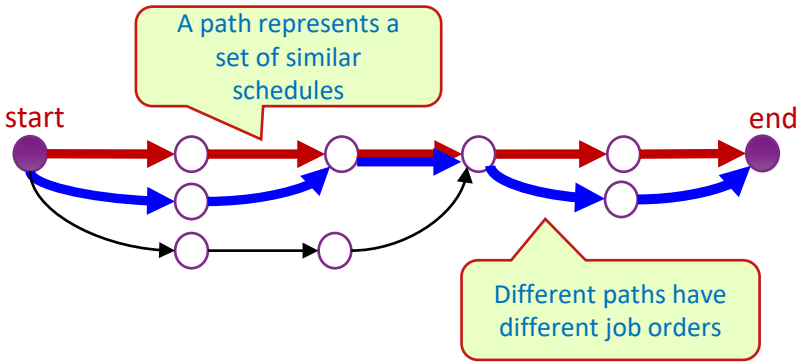
It is called the "schedule-abstraction graph"

Goal: an *accurate* and *efficient* analysis



Response-time analysis using schedule-abstraction graphs

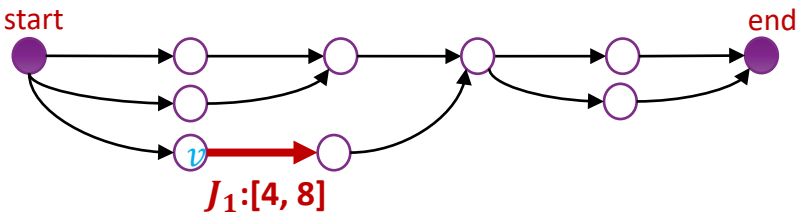
A path aggregates all schedules with the same job ordering



Response-time analysis using schedule-abstraction graphs

A **path** aggregates all schedules with the same job ordering

A **vertex** abstracts a system state and an **edge** represents a dispatched job

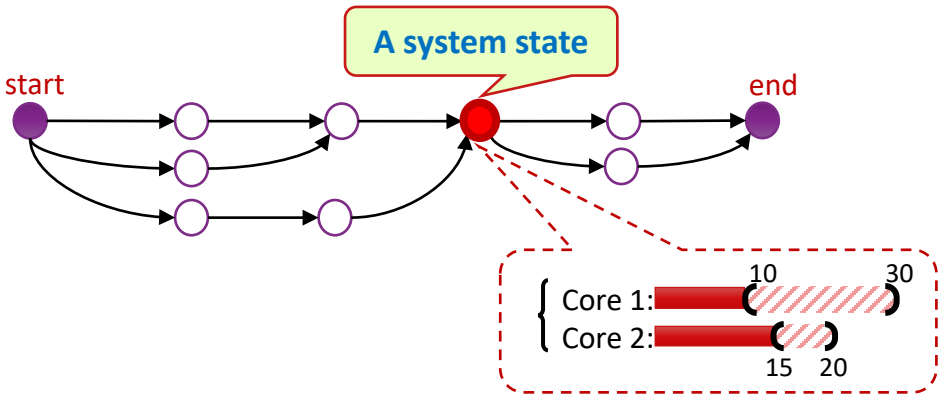


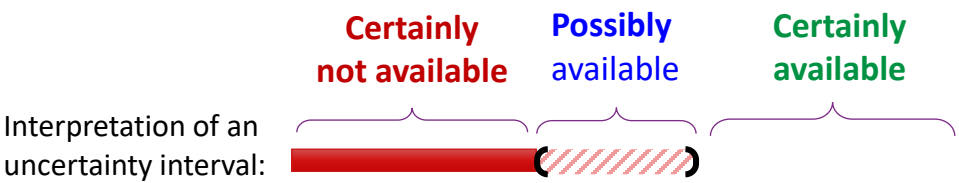
Response-time analysis using schedule-abstraction graphs

A path aggregates all schedules with the same job ordering

A vertex abstracts a system state and an edge represents a dispatched job

A state is labeled with the finish-time interval of any path reaching the state





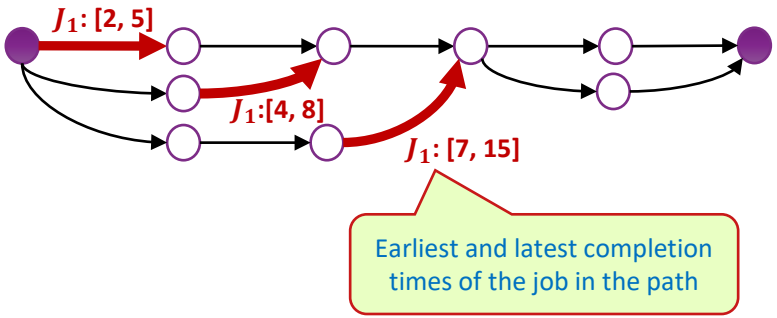
Response-time analysis using schedule-abstraction graphs

A **path** aggregates all schedules with the same job ordering

A **vertex** abstracts a system state and an **edge** represents a dispatched job

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

Obtaining the response time:

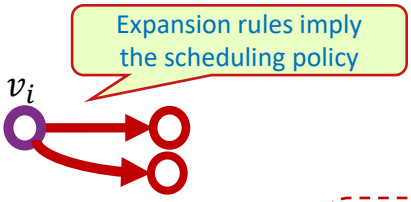


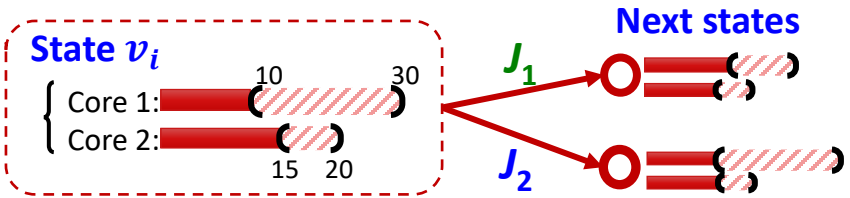
Best-case response time = **min** {completion times of the job} = **2**

Worst-case response time = **max** {completion times of the job} = **15**

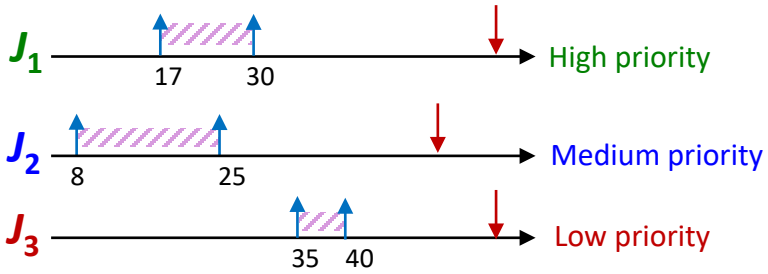
Building the schedule-abstraction graph

Expanding a vertex:
(reasoning on uncertainty intervals)





Available jobs
(at the state)



[ECRTS'2018]

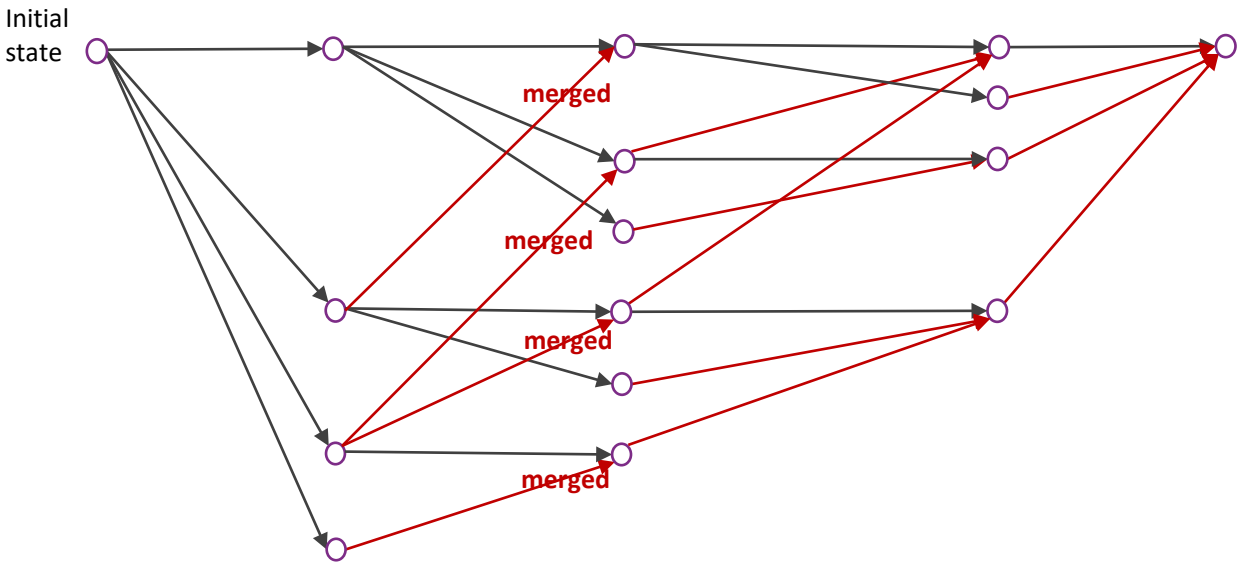
Building the schedule-abstraction graph

Building the graph
(a breadth-first method)

Repeat until every path includes all jobs

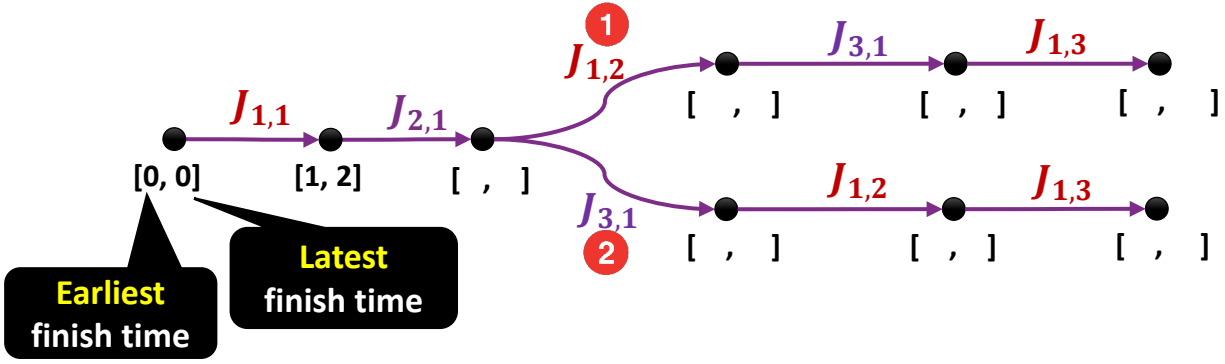
1. **Find the shortest path**
2. **For each not-yet-dispatched job that can be dispatched after the path:**
   - **2.1. Expand** (add a new vertex)
   - **2.2. Merge** (if possible, merge the new vertex with an existing vertex)

System is idle and
no job has been scheduled

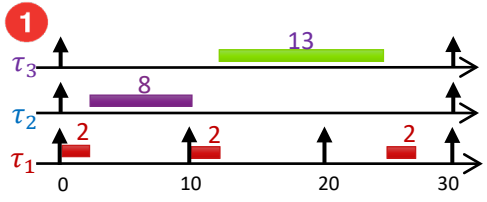


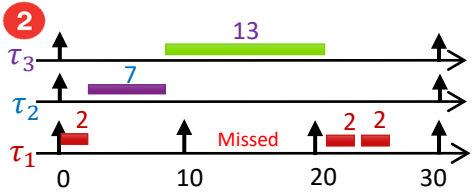
Abstracting schedules in a graph of job orderings

Task	Period	Execution time Min	Max	Jitter	
τ_3	30	[3,	13]	0	Low-priority
τ_2	30	[7,	8]	0	Medium-priority
τ_1	10	[1,	2]	0	High-priority



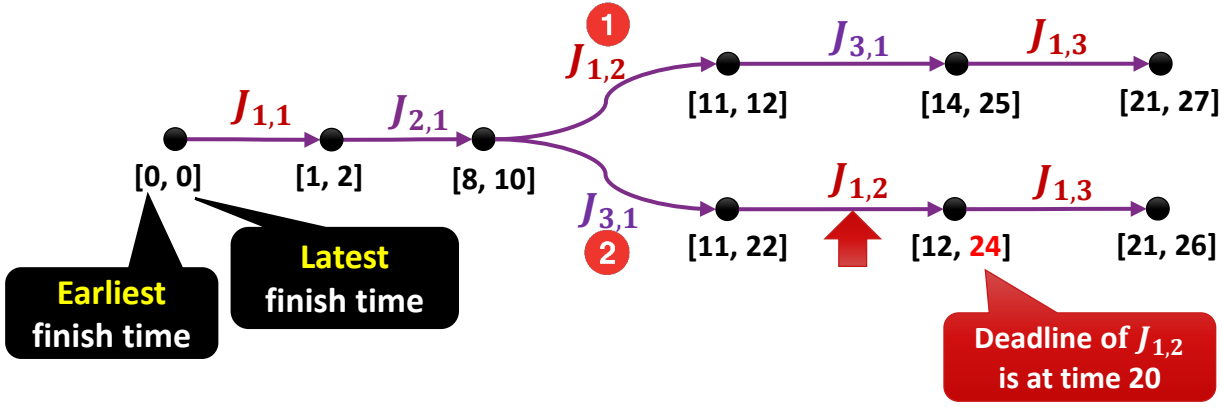
Each path shows a job ordering



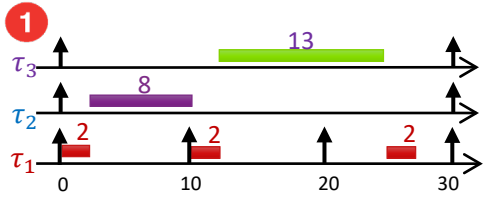


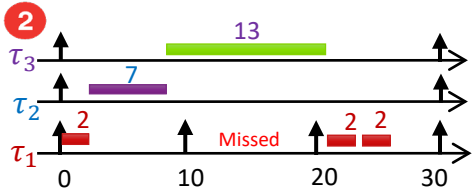
Abstracting schedules in a graph of job orderings

Task	Period	Execution time Min	Max	Jitter	
τ_3	30	[3,	13]	0	Low-priority
τ_2	30	[7,	8]	0	Medium-priority
τ_1	10	[1,	2]	0	High-priority

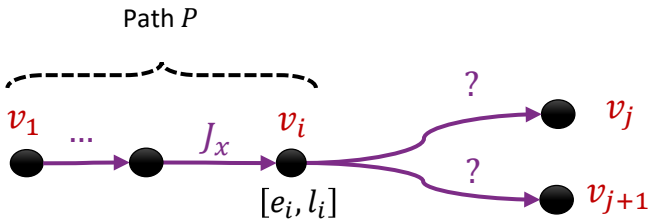


Each path shows a job ordering





Rules to generate a schedule-abstraction graph



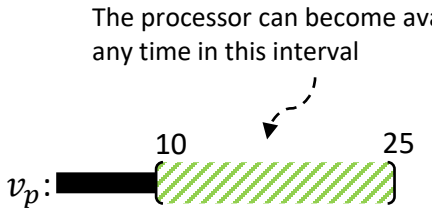
Given

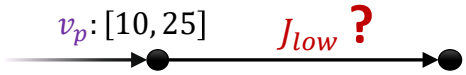
- a path P of edges that connect v_1 to v_p, and
- a set of jobs $\mathcal{J}$ that have not yet been dispatched on the path,

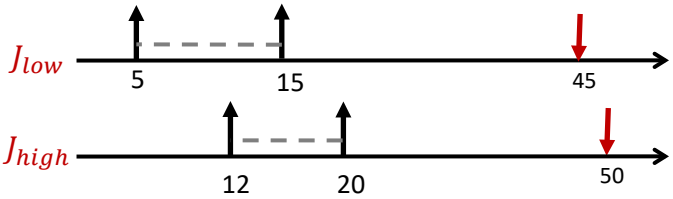
Find

- the set of jobs that can be the direct successors of state v_i
- For each such job, find the earliest and latest finish time of the job when it is dispatched directly after v_i

Deriving rules for expanding the graph





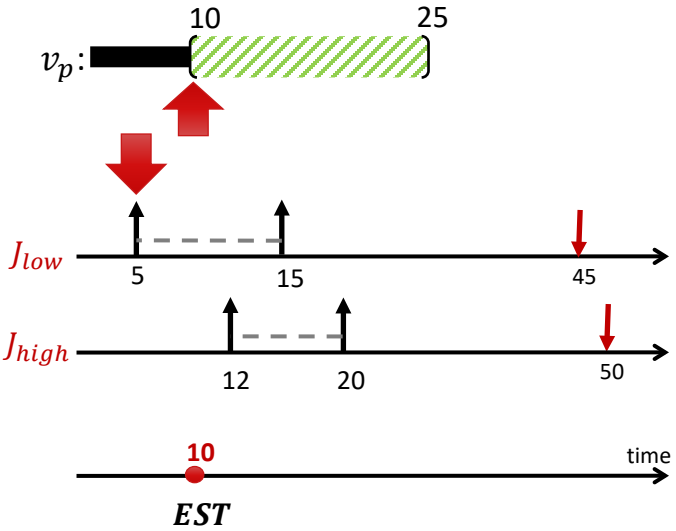


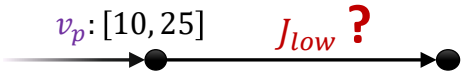
Q1: Can J_{low} be the first job that is being scheduled next?

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

Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_{low}	5	15	50	2	15	low
J_{high}	12	20	45	1	10	high

Deriving rules for expanding the graph





1

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

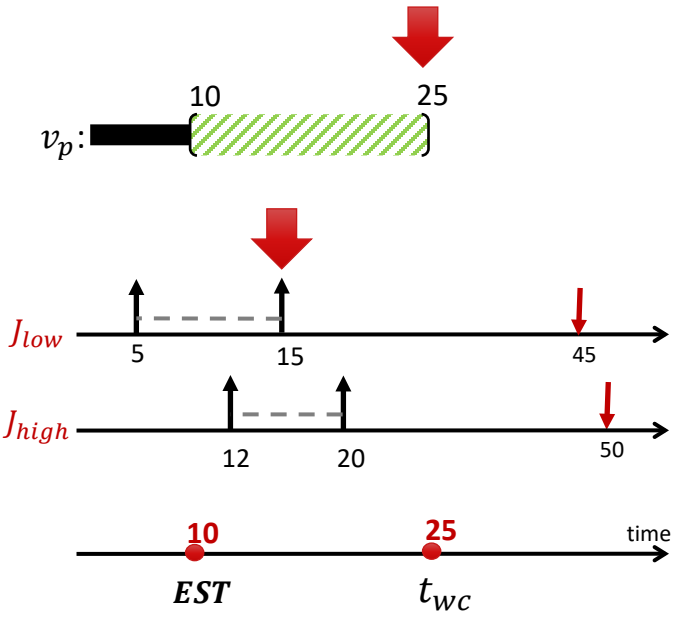
Try to imagine the best scenario that can happen for the start time of J_{low}

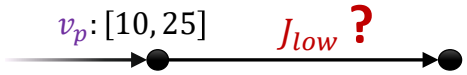
EST is the **earliest** time at which the job is **possibly released** and the processor is **possibly available**

EST = max { 5, 10 } = 10

Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_{low}	5	15	50	2	15	low
J_{high}	12	20	45	1	10	high

Deriving rules for expanding the graph





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** scheduling policy

1. Work-conserving rule

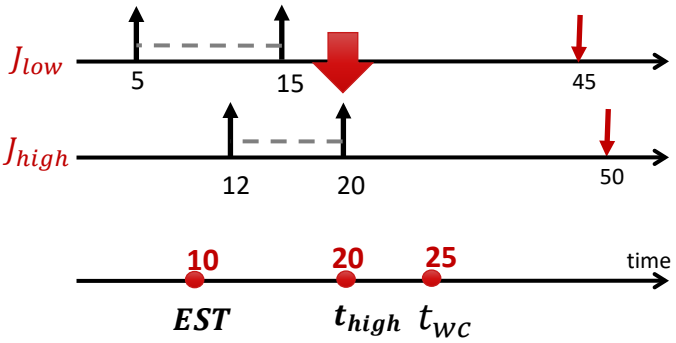
t_{wc} is the time at which **a job** is **certainly released** and the processor is **certainly available**

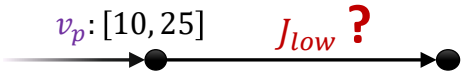
t_{wc} = max { 15, 25} = 25

Try to imagine a scenario in which there will be a certainly released job and a certainly available processor.

Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_{low}	5	15	50	2	15	low
J_{high}	12	20	45	1	10	high

Deriving rules for expanding the graph





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

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

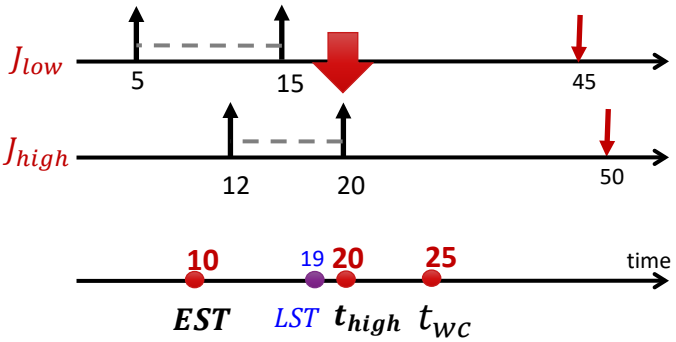
2. JLFP rule

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

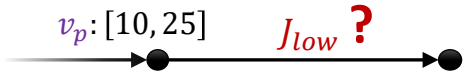
t_{high} = 20

Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_{low}	5	15	50	2	15	low
J_{high}	12	20	45	1	10	high

Deriving rules for expanding the graph



Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_{low}	5	15	50	2	15	low
J_{high}	12	20	45	1	10	high



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

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

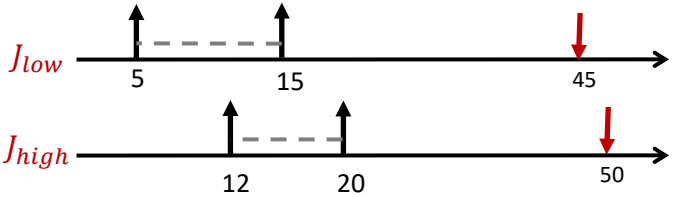
t_{high} is the time at which a higher-priority job is certainly released

$$t_{high} = 20$$

3. Latest start time (LST)

$$LST = \min\{t_{high} - 1, t_{wc}\}$$
$$= \min\{20 - 1, 25\} = 19$$

Deriving rules for expanding the graph



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 **EST** $\leq$ **LST** then add an edge for job J_{low}

Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_{low}	5	15	50	2	15	low
J_{high}	12	20	45	1	10	high

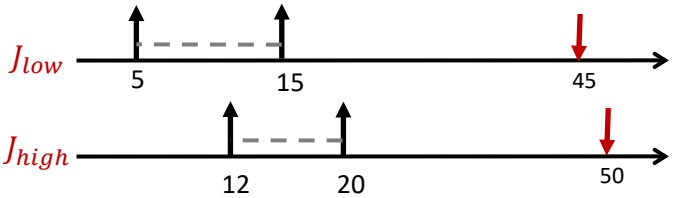
Earliest finish time (EFT) = EST + BCET

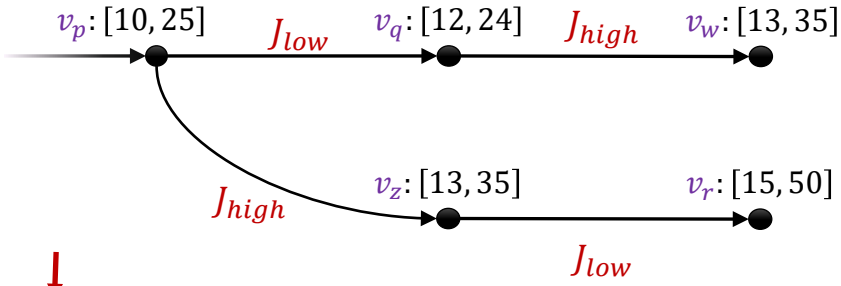
EFT = 10 + 2 = 12

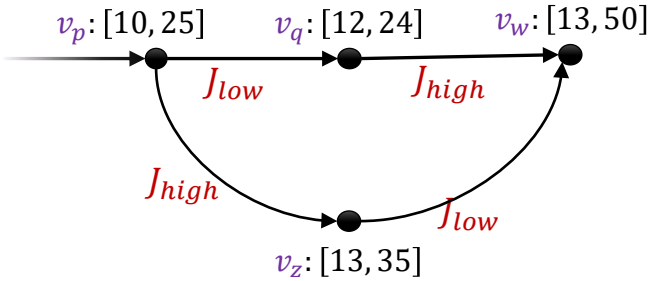
Latest finish time (LFT) = LST + WCET

LFT = 19 + 15 = 24

Complete graph



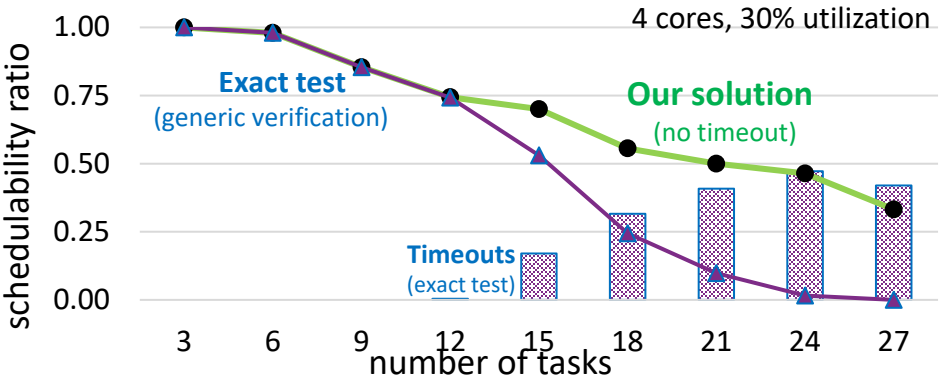




Job	Release time Min	Release time Max	Deadline	Execution time Min	Execution time Max	Priority
J_{low}	5	15	50	2	15	low
J_{high}	12	20	45	1	10	high

Taste of results

Comparison with an **exact schedulability test** implemented in UPPAAL, a **generic model-verification tool**



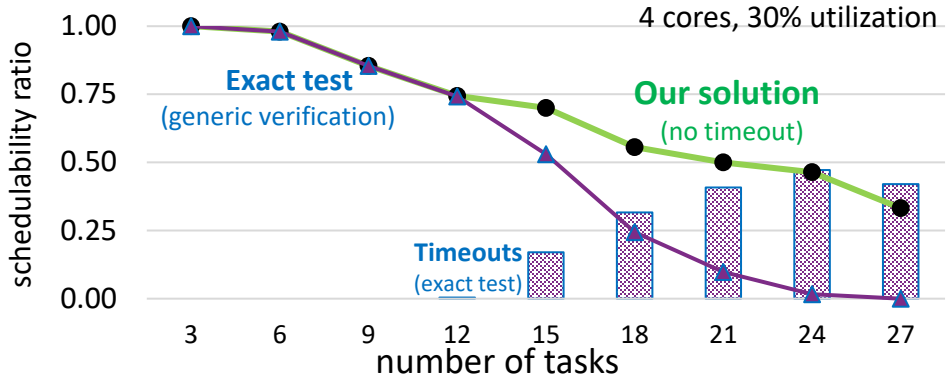
Schedulability = ratio of systems deemed schedulable to the total evaluated systems

[ECRTS'2019]

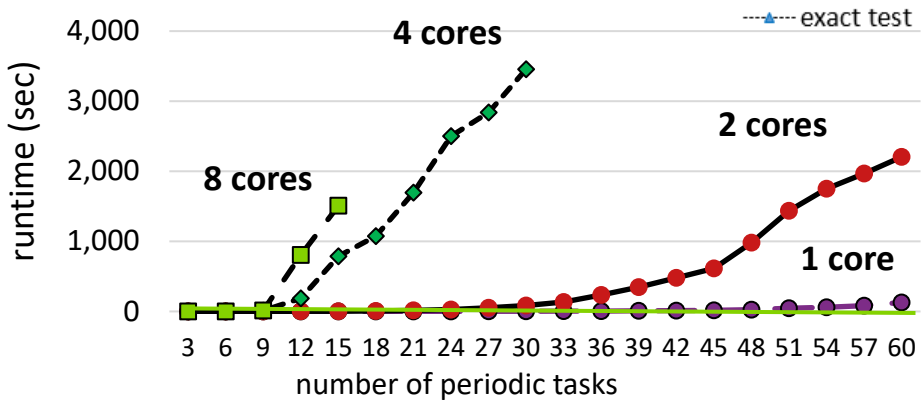
Taste of results

Comparison with an **exact schedulability test** implemented in UPPAAL, a **generic model-verification tool**

[ECRTS'2019]



Almost as accurate as the exact test



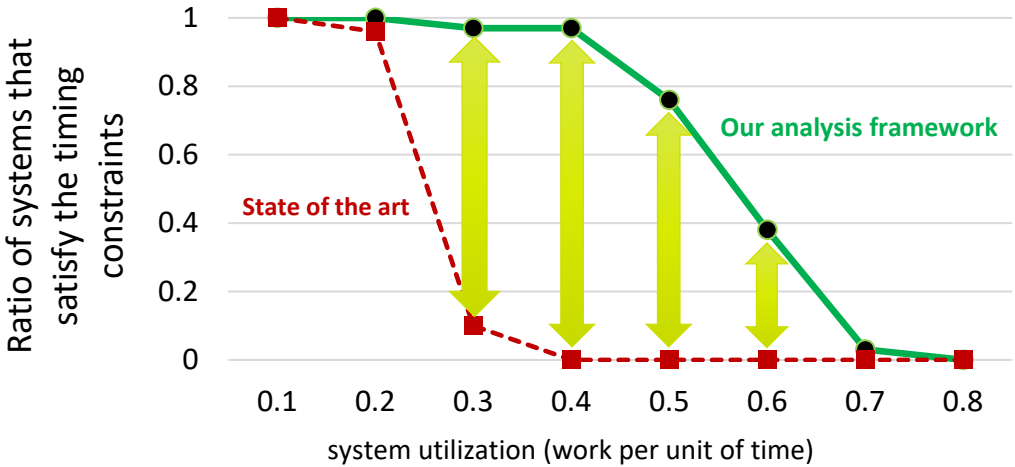
Much faster

this work (8 cores)

Schedulability = ratio of systems deemed schedulable to the total evaluated systems

Taste of results

Comparison against
closed-form analysis



Experiments:

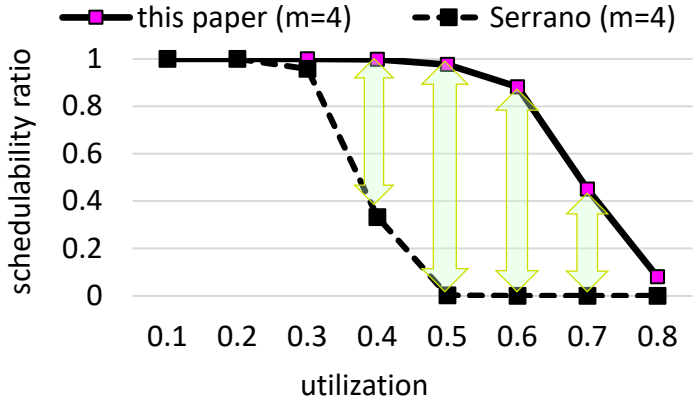
- A multiprocessor platform with 8 identical cores
- Each data point summarizes the results for 200 randomly generated task sets
- Each task set includes 10 parallel real-time tasks with periodic activations that have a directed-acyclic graph (DAG) data-flow dependency

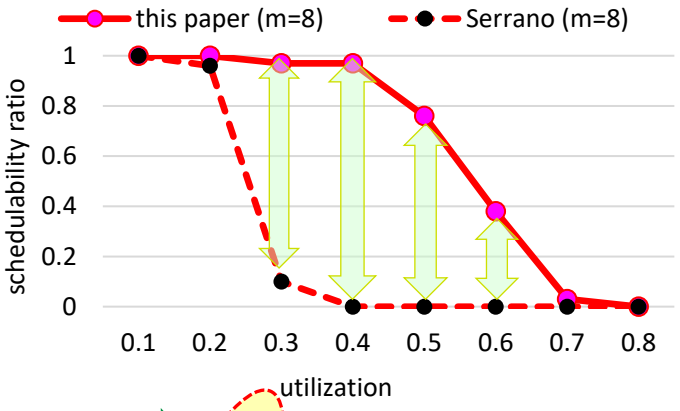
Schedulability = ratio of systems deemed schedulable to the total evaluated systems

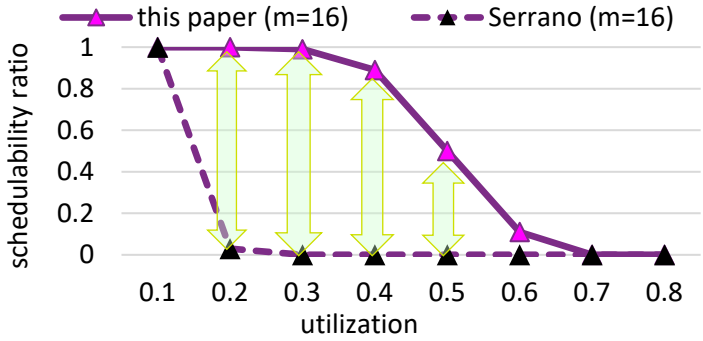
[ECRTS'2019]

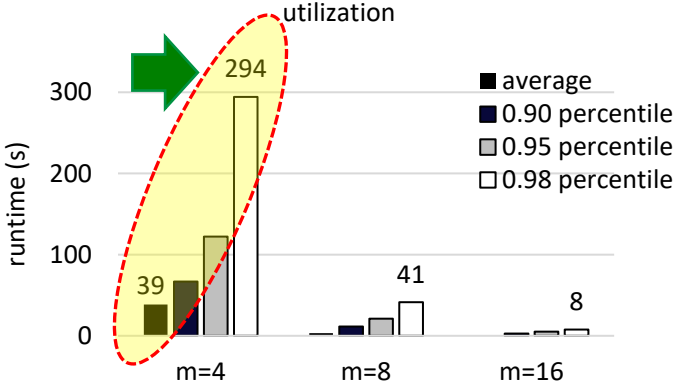
Parallel DAG tasks

10 parallel random DAG tasks









M. Serrano, A. Melani, S. Kehr, M. Bertogna, and E. Quiñones, "An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-Based Global Fixed Priority Scheduling", ISORC, 2017.

The journey: papers

Single core

- Independent jobs (with no precedence constraints)
- **Exact analysis** (tight bounds),

[RTSS'17]

Multicore

Independent jobs, sufficient analysis (close-to-tight bounds)

[ECRTS'18]

[ECRTS'19] Parallel (dependent) jobs, bounds are not tight

[DATE'19] Limited-preemptive tasks, tight bounds (UPPAAL) Intern student

[RTSS'20] Tasks that share data (real-time resource access policies, bounds are not tight) Master student

[ECRTS'22] Moldable Gang tasks Master student

[RTAS'22] Master student

Partial order reduction

(5 orders of magnitude faster)

The best paper award of RTAS'22

Distributed systems

[RTNS'22] Analyzing data-age for multi-rate DAG tasks with data dependencies

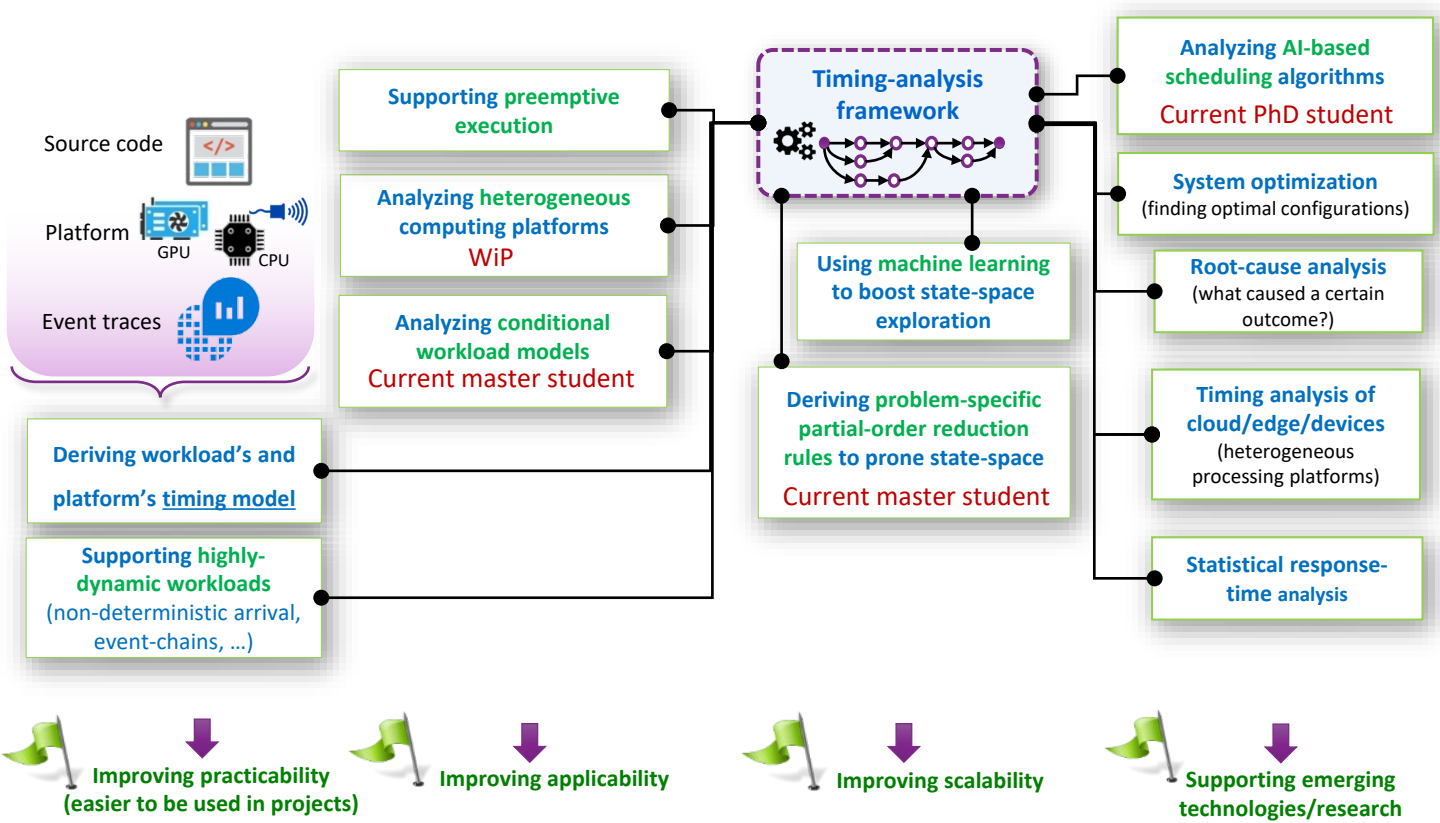
PhD student

The journey: current research

- Gang scheduling **(ECRTS'22)** — Master student (grade: 9)
- Supporting shared resources **(RTSS'20)** — Master student (9.5)
- Partial-order reduction (POR) **(RTAS'22)** — Master student (9)
- POR + precedence constraint **(journal extension of RTAS'22)**
- Preemptive scheduling **(to be submitted)** — Master student (8.5)
- Conditional DAGs (uncertainty in program paths) **(WiP)** — **Current master student**
- Finding counter examples **(WiP)** — **Current master student**
- End-to-end latency analysis in automotive applications **(RTNS'22)** — PhD student
- POR + multiprocessor platforms **(WiP)** — **Current master student**
- Analyzing tasks with core-affinities under reservation-based scheduling **(WiP)** — PhD student

Future of schedule-abstraction graph

Extensions of schedule-abstraction graph



Any questions?

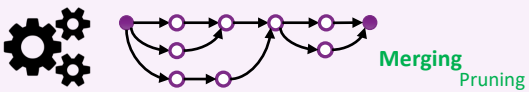
3000x faster than
generic timing verification tools (e.g., UPPAAL)

Workload model
(timing features of the workload)

Resource model

Scheduling algorithm
(dynamic and static scheduling algorithms)

Timing-analysis framework
(it is a formal verification engine dedicated to **timing models** and **timing properties**)



Worst-case Performance Bounds
(for para-functional properties)

- **Lower and upper bounds on**
  - response time, latency, throughput, jitter, quality of service/control
- **Counter examples** in case of timing violation