Partial-Order Reduction for Schedule-Abstraction-based Response-Time Analyses of Non-Preemptive Tasks

Sayra Ranjha

Geoffrey Nelissen

Mitra Nasri

Unfortunately, she could not present her work today.

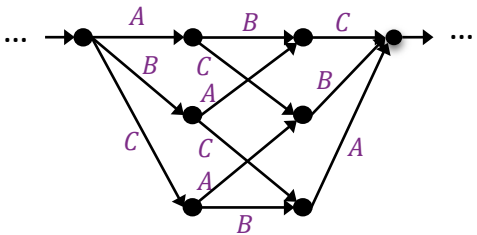
RTAS 2022

The paper in a nutshell

Schedule-abstraction graph (SAG) is a recent reachability-based technique for **response-time analysis**

- Highly accurate
- Extendible (supports parallel DAG tasks, multiprocessors, shared resources, …)
- At least **3000 times faster than UPPAAL**

However, it still suffers from state-space explosion:



The idea of partial-order reduction (POR)

Do not explore job schedules that have no impact on schedulability

Impact of our solution

Speedup: **5 orders of magnitude**

98% reduction in explored states

Scales to millions of jobs

Only with 0.1% increase on the response-time estimates

Publications on SAG:
[RTSS'17, ECRTS'18, ECRTS'19, DATE'19, RTSS'20, RTSS'21, RTNS'22, …]

Response-time analysis: where are we?

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

- Fast

- Pessimistic
- Hard to extend

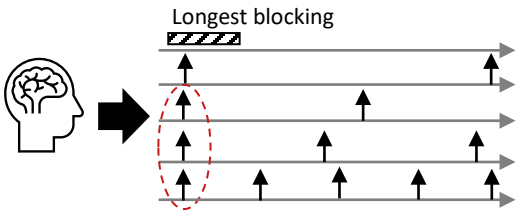
$$R_i^{(0)} = C_i + \sum_{j=1}^{i-1} C_j$$

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

priority tasks. A response-time
k-set with a limited-preemptive
er is computed by iterating the
ed point is reached, starting with

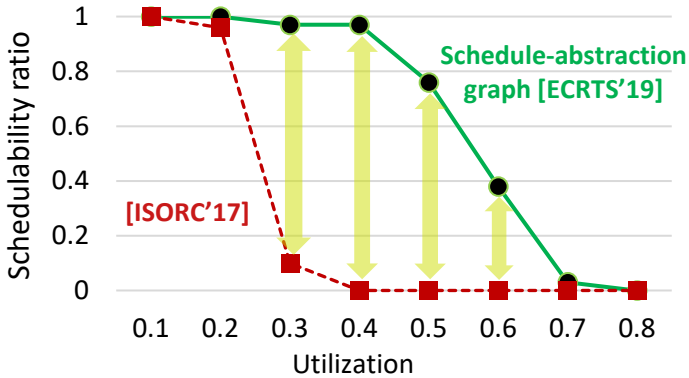
$R_k = len(G_k) + \frac{1}{m}(vol(G_k) - len(G_k))$:

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



What is the current status?

Experiment: limited-preemptive DAG scheduling
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)

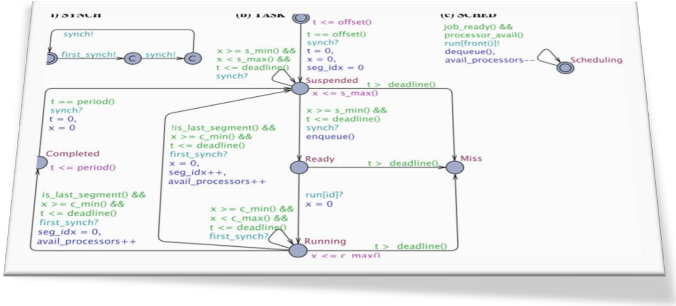
- Fast

- Pessimistic
- Hard to extend

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

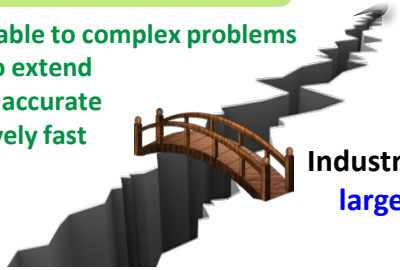
- Accurate
- Easy to extend

- 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, …]

Schedule-abstraction graph in a nutshell

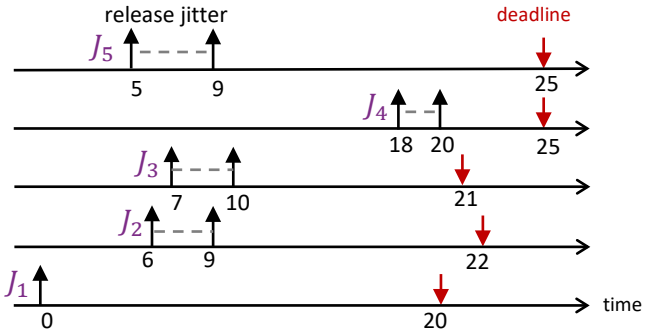
Define an **observation window**
(which includes the worst-case response time of the tasks)

For synchronous periodic tasks, this could be their **hyperperiod**

Construct a **job set** (i.e., jobs that can possibly be released in that window)

Job set

Job	Release time Earliest	Release time Latest	BCET	WCET	Deadline	Priority
J_1	0	0	7	13	20	1 (highest)
J_2	6	9	1	2	22	2
J_3	7	10	1	2	21	3
J_4	18	20	1	3	25	4
J_5	5	9	1	2	25	5 (lowest)



Resource model

Number of cores

Scheduling policy

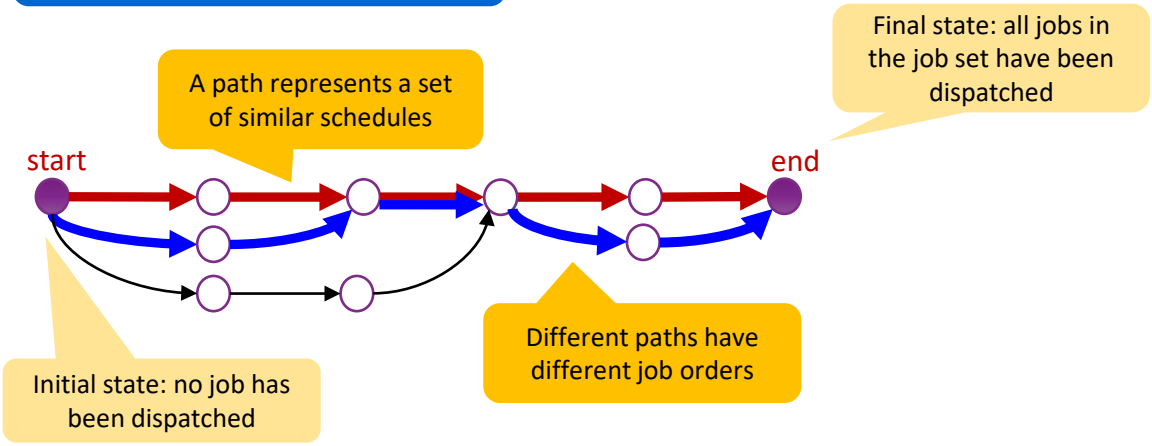
Non-preemptive

Job-level fixed-priority (JLFP)

Examples: EDF, FP, etc.

Response-time analysis using schedule-abstraction graph technique

A path aggregates all schedules with the same job ordering



Main idea:

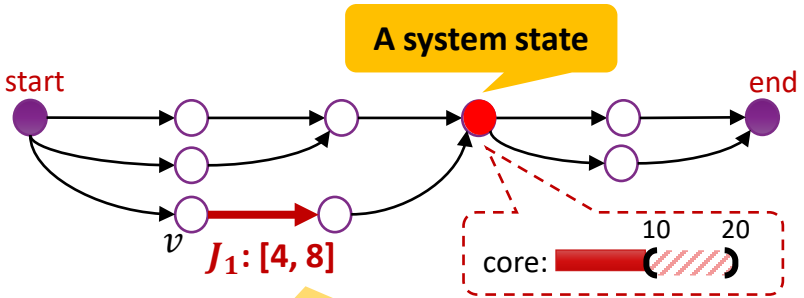
Use job-ordering abstraction to build a graph that abstracts all possible schedules that can happen for a given job set when scheduled by a given scheduling policy

Response-time analysis using schedule-abstraction graph technique

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



Earliest and latest finish time of J_1 when it is dispatched after state v

Interpretation of an uncertainty interval:

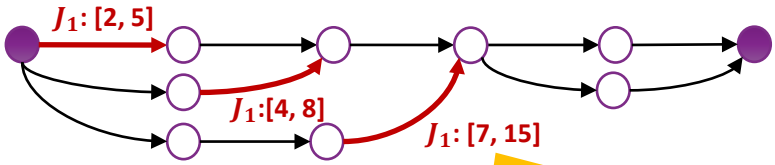
Response-time analysis using schedule-abstraction graph technique

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:



Earliest and latest completion times of the job in the path

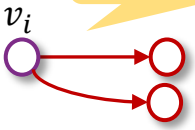
Best-case response time = **min** {completion times of the job} = **2**

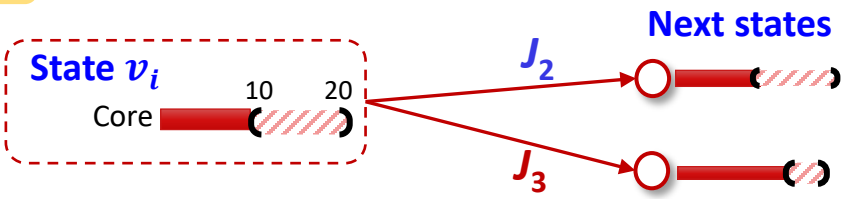
Worst-case response time = **max** {completion times of the job} = **15**

Constructing the schedule-abstraction graph

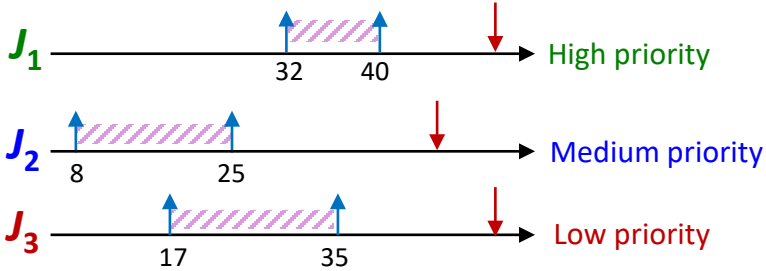
Expanding a vertex
(reasoning on uncertainty intervals)

Expansion rules imply the scheduling policy





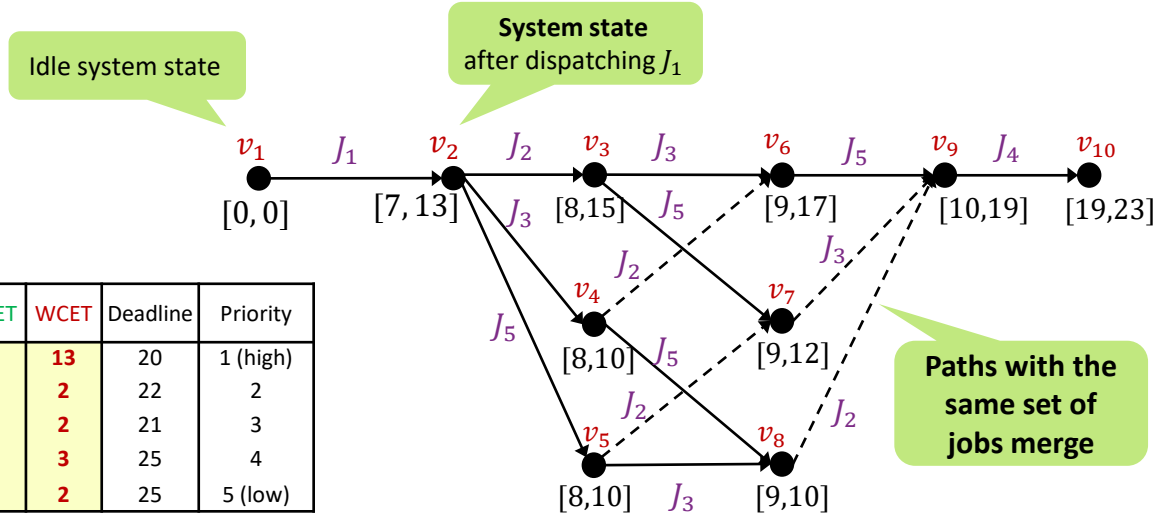
Not-yet-dispatched (available) jobs
(at the state v_i)



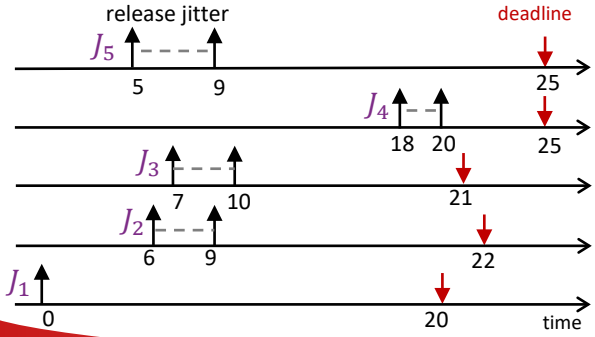
Scalability challenge of the schedule-abstraction graph technique

Schedule-abstraction graph: challenges

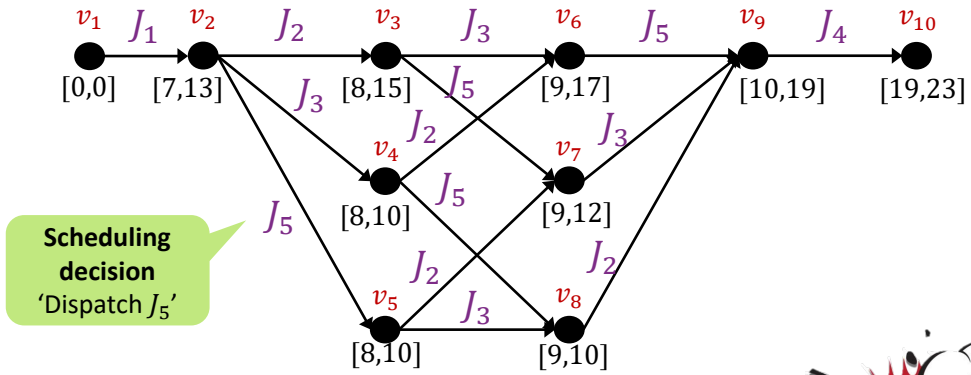
An example



Job	Release time Earliest	Release time Latest	BCET	WCET	Deadline	Priority
J_1	0	0	7	13	20	1 (high)
J_2	6	9	1	2	22	2
J_3	7	10	1	2	21	3
J_4	18	20	1	3	25	4
J_5	5	9	1	2	25	5 (low)



Schedule-abstraction graph: challenges



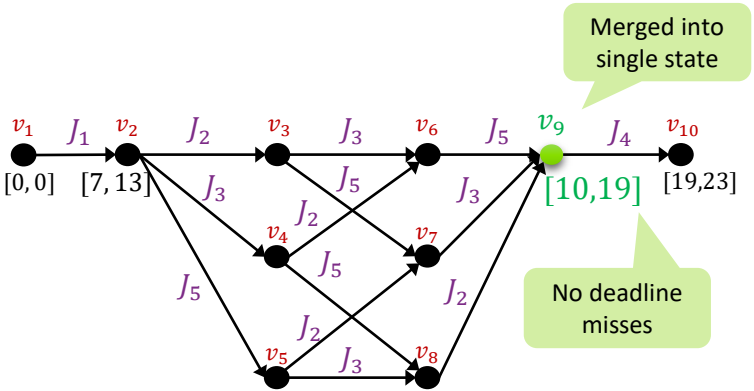
Each edge can only account for a **single scheduling decision**

→

Combinatorial exploration in case of **timing uncertainties**

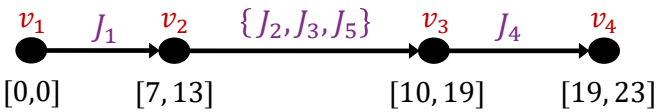
→

Partial-order reduction: why?



Combinatorial exploration of $\{J_2, J_3, J_5\}$

Not relevant to asserting schedulability



Scheduling decision
'Dispatch $\{J_2, J_3, J_5\}$'

Use **partial-order reduction** to avoid combinatorial exploration

Deadlines: 20, 21, 22, 25

Agenda

- Why schedule-abstraction graph?
- Scalability challenge of the schedule-abstraction graph technique
- **Partial-order reduction**
  - **How to find reduction sets**
  - **How to estimate response-time bounds**
- Evaluations
- Conclusions and future work

Partial-order reduction: what?

Research question:
How to avoid exploring job schedules that have no impact on schedulability?

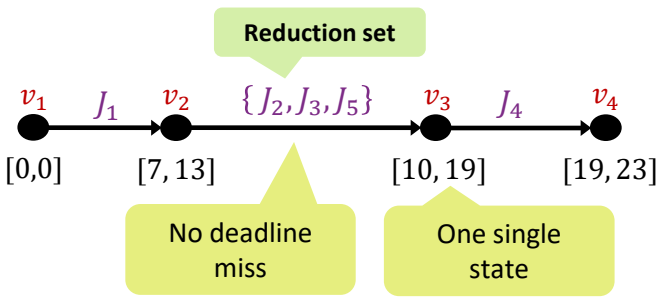
While having

an exact schedulability analysis + **with relatively low pessimism on the response time bounds**

Exploring job schedules is irrelevant when

all those schedules lead to the **same system state**

none of the jobs observe a deadline miss

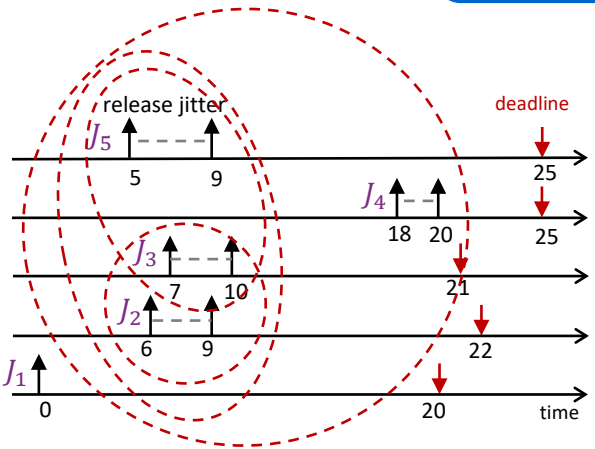


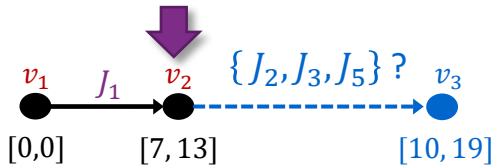
Partial-order reduction: how?

Before exploring a state:

Form a candidate **reduction set** → Derive **response time bounds** for the jobs in the **reduction set**

- **No deadline miss** → **Reduce** the set to a **single edge** and **form the new state**
- **Potential deadline miss** → Let the **original schedule-abstraction graph** explore all scheduling decisions





How to form such a candidate **reduction set**?

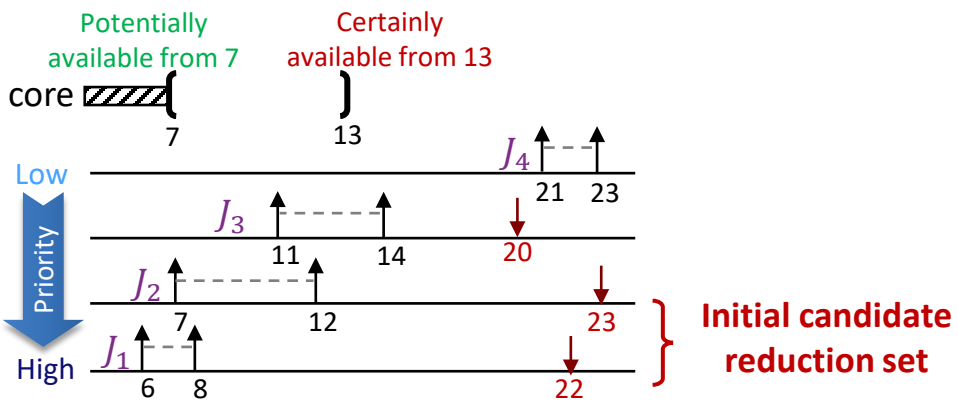
How to quickly derive the **response-time bounds** of each job and assert its **schedulability**?

Forming a candidate reduction set

A set of jobs is a **safe reduction set** (denoted by J^M) iff no other not-yet-dispatched job that does not belong to J^M can **interfere** with them (namely, start executing before jobs in J^M complete their execution).

Step 1: include all jobs that can potentially execute next

How? Use the original schedule-abstraction graph rules to obtain them (polynomial time)



Forming a candidate reduction set

Step 2: check whether there are other jobs that can **potentially interfere** with the reduction set J^M

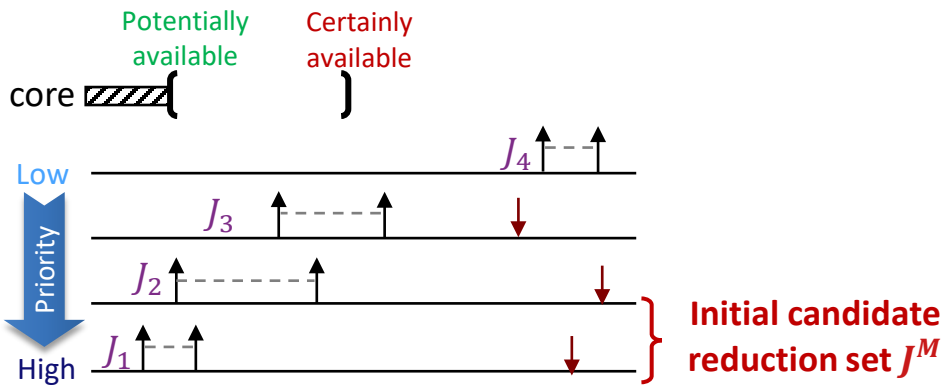
Step 3: if yes, check if they can be added to the reduction set

Case 1: work-conserving scheduling property

A job **interferes** with a reduction set if it is released <u>before or during</u> an **idle interval** between jobs in the reduction set J^M

Case 2: priority-driven scheduling property

A job **interferes** with a reduction set if it is released before a **lower-priority** job in the reduction set J^M executes



Forming a candidate reduction set

Case 1: work-conserving scheduling property

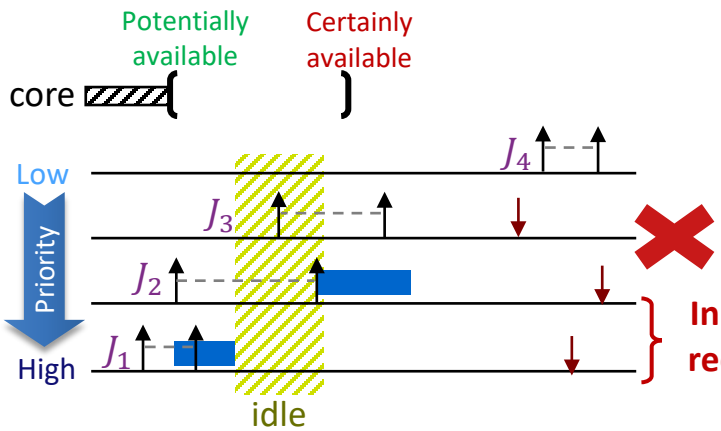
A job **interferes** with a reduction set if it is released before or during an **idle interval** between jobs in the reduction set J^M

Idea

If the **earliest finish time (EFT)** of the jobs that certainly release before time t is smaller than t, there might be an idle time just before t.

There are at most $|J^M| - 1$ idle intervals.

See Equation (2) and Lemmas 1-4 in the paper



J_3 interferes with the candidate reduction set $\{J_1, J_2\}$

Initial candidate reduction set J^M

Forming a candidate reduction set

Case 2: priority-driven scheduling property

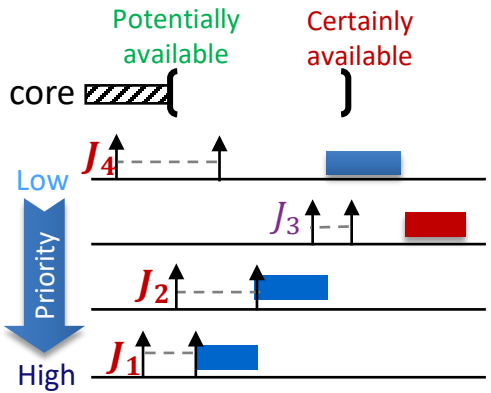
A job **interferes** with a reduction set if it is released before a **lower-priority** job in the reduction set J^M executes

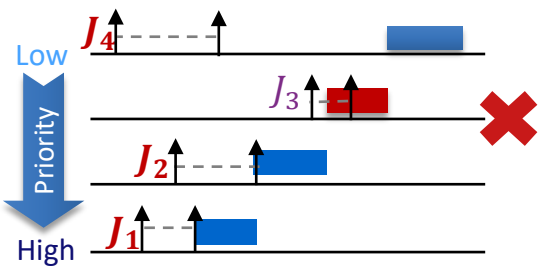
Idea

If the **latest start time (LST)** of any job in J^M is smaller than the release time of a higher-priority job that is NOT in J^M, **then** no higher-priority job can interfere with J^M

See Lemma 5 in the paper

How to obtain the **latest start time (LST)** of a job in J^M?





J_3 interferes with the candidate reduction set $\{J_1, J_2, J_4\}$

Candidate reduction set $J^M = \{J_1, J_2, J_4\}$

Checking schedulability of a reduction set

We derive **two upper bounds** on the **latest finish time (LFT)** of each job in the reduction set

→ **Upper bound from a sufficient fixed-point iteration test**

→ **Upper bound from a priority-agnostic simulation-based test**

The paper also explains how to obtain bounds on
- the **earliest** and **latest** **start time** (**EST** and **LST**) and
- the **earliest** and **latest** **finish time** (**EFT** and **LFT**)

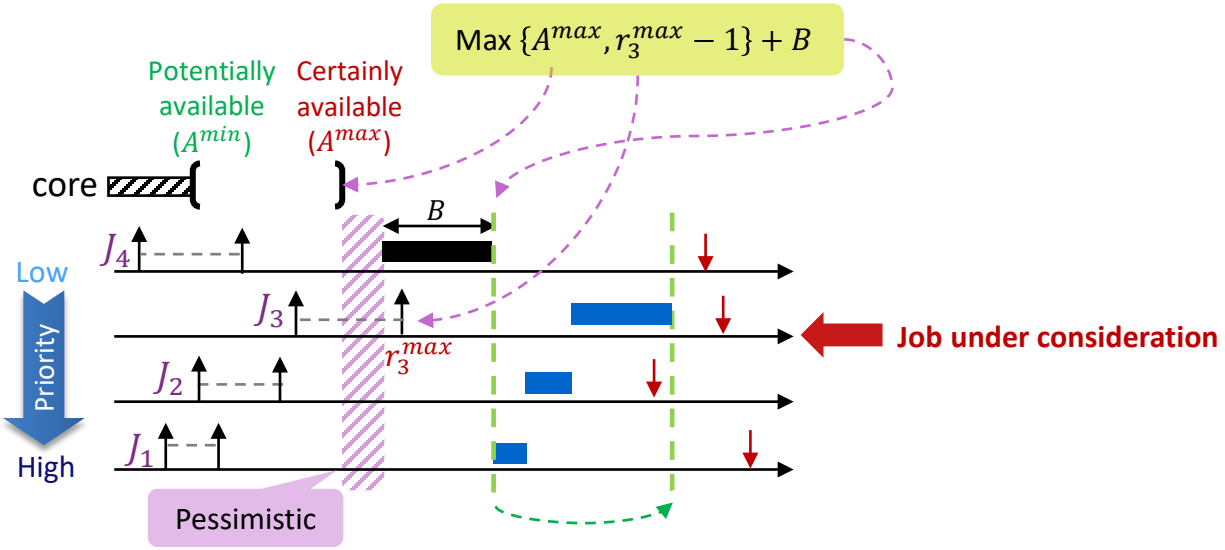
Upper bound from a sufficient fixed-point iteration test

For any job J_i in the reduction set J^M:

Step 1: Assume that the core stays busy for as long as possible

Step 2: find the **maximum blocking time (B)** from lower-priority jobs in J^M

Step 3: use a fixed-point iteration to find the **latest finish-time** of the job J_i assuming that "the higher-priority jobs in J^M that could execute before J_i finishes, will execute for **as long as possible**"

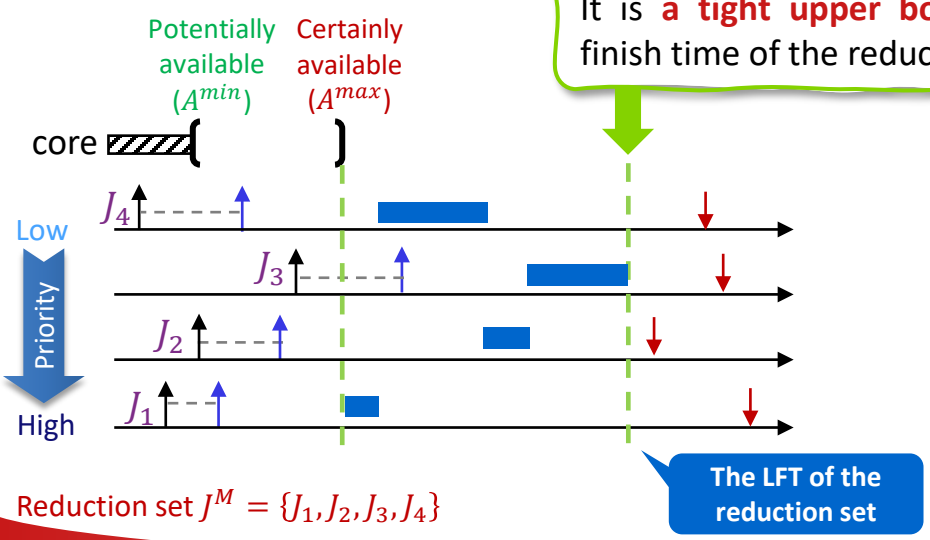


Reduction set $J^M = \{J_1, J_2, J_3, J_4\}$, job under consideration: J_3

Upper bound from a priority-agnostic simulation-based test

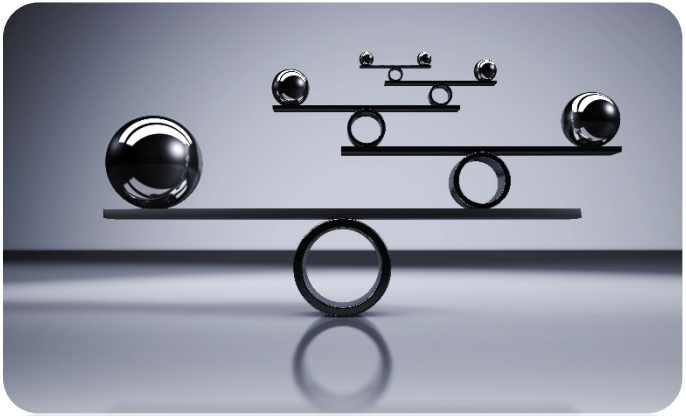
Step 1: assume that the core stays busy for as long as possible
Step 2: sort the jobs by their latest release time r^{max}
Step 3: calculate the **latest finish time (LFT)** of each job by taking the maximum between the LFT of the previous job and the latest release time (r^{max}) of the job. Add WCET of the job to it.

It is **a tight upper bound** on the latest-finish time of the reduction set. [Lemma 8]



Reduction set $J^M = \{J_1, J_2, J_3, J_4\}$

Agenda



- Why SAG?
- Challenges of SAG
- Partial-order reduction
  - How to find reduction sets
  - How to estimate response-time bounds

Evaluations

- Conclusions and future work

Empirical experiments

Does partial-order reduction (POR) provide a **speedup** and **state-space reduction** over the original SAG implementation?

How is the **worst-case response time** (WCRT) bound affected by the partial-order reduction?

Speedup | **State-space reduction** | **Accuracy** | **Scalability**

Case study

(**Automotive use case** from the WATERS 2017 industrial challenge)

Synthetic task sets

Case study: automotive task sets

APP4MC specification of the automotive use case from the WATERS 2017 industrial challenge [Hamann'17]

710 periodic runnables
Or **7 periodic tasks**

Preemption	Clock speed	#tasks/ runnables	#jobs per hyperperiod	Jitter	CPU time (s)		#states		Average #jobs per edge	Speedup	State-reduction ratio (%)
					Original	POR	Original	POR			
Task-level	1.2 GHz ($U \approx 0.18$)	7	786	No	0.00538	0.00574	809	766	1.027	0.937	5.315
				Yes	0.00951	0.00441	1248	586	1.344	2.157	53.045
Runnable-level	400 MHz ($U \approx 0.53$)	710	37,459	No	782.044	20.968	1,076,504	15,261	2.455	37.298	98.582
				Yes	Out of memory	13.996	N/A	351	107.026	∞	∞

With the provided clock speed (200 MHz), total utilization was larger than 1. We assumed first clock speeds that resulted in schedulable task sets.

A. Hamann, D. Dasari, S. Kramer, et al., "WATERS Industrial Challenge 2017," WATERS, 2017.

Case study: automotive task sets

APP4MC specification of the automotive use case from the WATERS 2017 industrial challenge [Hamann'17]

The **overhead** of POR may result to no gain in runtime when the **problem is too simple and small**

Preemption	Clock speed	#tasks/ runnables	#jobs per hyperperiod	Jitter	CPU time (s)		#states		Average #jobs per edge	Speedup	State-reduction ratio (%)
					Original	POR	Original	POR			
Task-level	1.2 GHz ($U \approx 0.18$)	7	786	No	0.00538	0.00574	809	766	1.027	**0.937**	5.315
				Yes	0.00951	0.00441	1248	586	1.344	**2.157**	53.045
Runnable-level	400 MHz ($U \approx 0.53$)	710	37,459	No	782.044	20.968	1,076,504	15,261	2.455	**37.298**	98.582
				Yes	Out of memory	13.996	N/A	351	107.026	∞	∞

POR allows SAG to scale to much larger problems with more uncertainty

The larger the problem, the greater the impact of POR!

$$\textbf{Speedup} = \frac{\text{Runtime of the } \textbf{original SAG}}{\text{Runtime of } \textbf{SAG + POR}}$$

higher

A. Hamann, D. Dasari, S. Kramer, et al., "WATERS Industrial Challenge 2017," WATERS, 2017.

Case study: automotive task sets

APP4MC specification of the automotive use case from the WATERS 2017 industrial challenge [Hamann'17]

Considerable **gain** in reducing the number of states

Preemption	Clock speed	#tasks/ runnables	#jobs per hyperperiod	Jitter	CPU time (s)		#states		Average #jobs per edge	Sp	State-reduction ratio (%)
					Original	POR	Original	POR			
Task-level	1.2 GHz ($U \approx 0.18$)	7	786	No	0.00538	0.00574	809	766	1.027	0.937	5.315
				Yes	0.00951	0.00441	1248	586	1.344	2.157	53.045
Runnable-level	400 MHz ($U \approx 0.53$)	710	37,459	No	782.044	20.968	1,076,504	15,261	2.455	37.298	98.582
				Yes	Out of memory	13.996	N/A	351	107.026	∞	∞

We reduced the number of states by **a factor of 70**.

When there is jitter, POR can find larger reduction sets!

State-reduction ratio $= 1 - \frac{\text{States explored by SAG + POR}}{\text{States explored by original SAG}}$ **higher**

A. Hamann, D. Dasari, S. Kramer, et al., "WATERS Industrial Challenge 2017," WATERS, 2017.

Synthetic task sets for the scalability experiment

- Periods drawn from **log-uniform** distribution in range [10, 100] ms
- Release jitter: 100 µs
- Execution time variation: 100% (BCET = 0)
- Utilization: 30%
- Timeout of 4 hours

Comparison

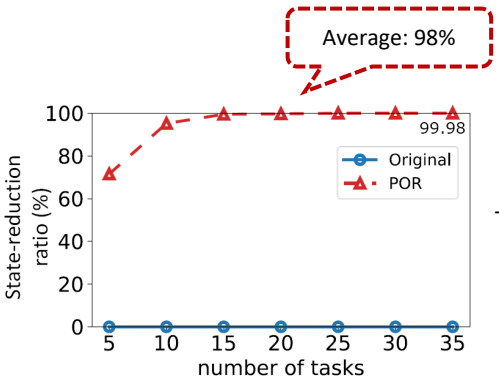
- Tasks: {5, 10, 15, 20, 25, 30, 35}
- Max jobs per hyperperiod: **50,000**

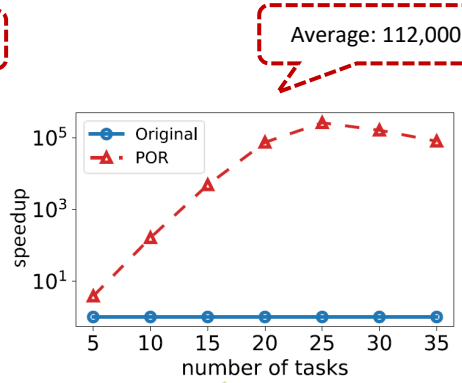
Scalability experiment

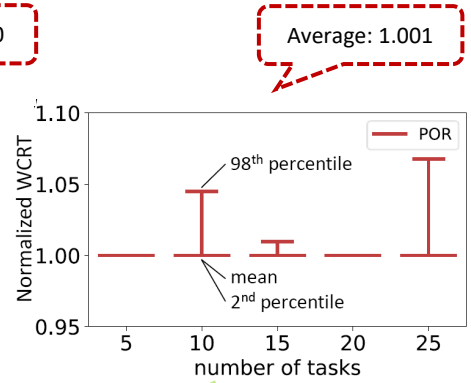
- Tasks: {10, 20, 30, **40, 50, 60, 70**}
- Max jobs per hyperperiod: **6,000,000**

Comparing with the original SAG

State-space reduction, speedup, accuracy







A significant reduction in the number of states

5 orders of magnitude faster

Negligible impact on the WCRT estimate

POR finds larger reduction sets when: there are more tasks or larger jitter

higher

$$\frac{\text{WCRT reported by SAG + POR}}{\text{WCRT reported by original SAG}}$$

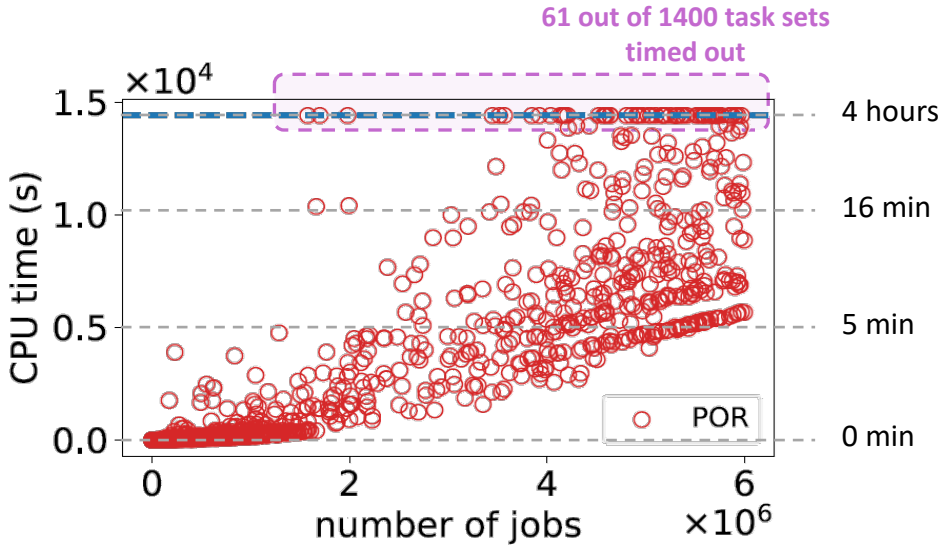
higher

$$1 - \frac{\text{States explored by SAG + POR}}{\text{States explored by original SAG}}$$

higher

$$\frac{\text{Runtime of the original SAG}}{\text{Runtime of SAG + POR}}$$

Scalability of partial-order reduction

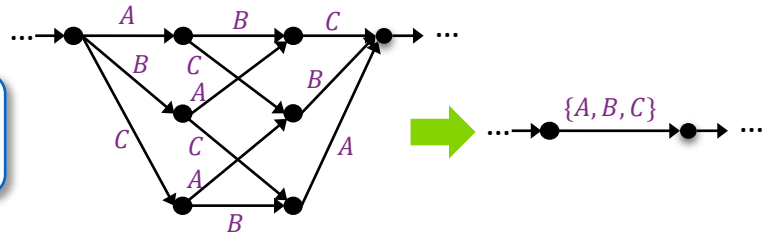


We scale to **millions** of jobs instead of tens of thousands

Summary and conclusions

Key idea

Do not explore job schedules that have no impact on schedulability



We proposed techniques to

Form safe reduction sets

Derive response time bounds for the jobs in a reduction set

Impact

POR becomes more efficient when the release jitter or the number of tasks increases!

Speedup: **5 orders of magnitude** on average

State-space reduction: **98%** on average

Scalability: **millions** of jobs instead of tens of thousands

Over-approximation of the WCRT: **0.1%** on average

Future work on partial-order reduction

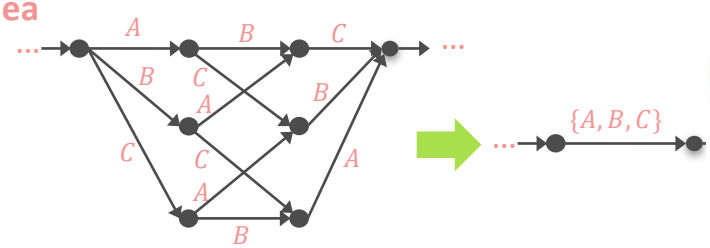
Extensions to tasks with dependencies (precedence constraints)

Extensions to multiprocessor platforms and parallel tasks

Deriving systematic rules to incorporate partial-order reduction to the future extensions of SAG

Any question?

Key idea



We proposed techniques to

Form safe reduction sets

Derive response time bounds for the jobs in a reduction set

Impact

Speedup: **5 orders of magnitude** on average

State-space reduction: **98%** on average

Scalability: **millions** of jobs instead of tens of thousands

Over-approximation of the WCRT: **0.1%** on average

Code: https://github.com/gnelissen/np-schedulability-analysis

Thank you …