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

Unfortunately, she could not present her work today.

Sayra Ranjha
Geoffrey Nelissen
Mitra Nasri

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, ...]
Partial-order reduction for schedule-abstraction-based response-time analysis

Response-time analysis: where are we?

Closed-form analyses (e.g., problem-window analysis)
- Fast
- Pessimistic
- Hard to extend

Closed-form analysis:
\[ R^{(0)}_i = C_i + \sum_{j=1}^{i-1} C_j \]
\[ R^{(k)}_i = C_i + \sum_{j=1}^{i-1} \left( \frac{R^{(k-1)}_j}{T_j} \right) C_j \]

- Experiment: limited-preemptive DAG scheduling
- Setup: 8 cores, 10 parallel DAG tasks

What is the current status?


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

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

**Job set**

<table>
<thead>
<tr>
<th>Job</th>
<th>Release time</th>
<th>BCET</th>
<th>WCET</th>
<th>Deadline</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>0</td>
<td>7</td>
<td>13</td>
<td>20</td>
<td>1 (highest)</td>
</tr>
<tr>
<td>$J_2$</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>$J_3$</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>$J_4$</td>
<td>18</td>
<td>1</td>
<td>3</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>$J_5$</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>25</td>
<td>5 (lowest)</td>
</tr>
</tbody>
</table>

**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.

A path represents a set of similar schedules.

Initial state: no job has been dispatched.

Different paths have different job orders.

Final state: all jobs in the job set have been dispatched.

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.

A system state

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.

A system state

Early and latest finish time of $J_1$ when it is dispatched after state $v$.

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

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

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available

Response-time analysis using schedule-abstraction graph technique

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

Interpretation of an uncertainty interval:

- Certainly not available
- Possibly available
- Certainly available
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:

Best-case response time = \( \min \{ \text{completion times of the job} \} = 2 \)

Worst-case response time = \( \max \{ \text{completion times of the job} \} = 15 \)
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 $v_i$)

$J_1$ High priority

$J_2$ Medium priority

$J_3$ Low priority

Partial-order reduction for schedule-abstraction-based response-time analysis
Scalability challenge of the schedule-abstraction graph technique
Schedule-abstraction graph: challenges
An example

Idle system state
System state after dispatching $J_1$

Paths with the same set of jobs merge
Schedule-abstraction graph: challenges

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

Combinatorial exploration in case of timing uncertainties

State-space explosion

Scheduling decision ‘Dispatch $J_5$’

Partial-order reduction for schedule-abstraction-based response-time analysis
Partial-order reduction: why?

Use partial-order reduction to avoid combinatorial exploration

Merged into single state

No deadline misses

Scheduling decision ‘Dispatch \{J_2, J_3, J_5\}’

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

Not relevant to asserting schedulability

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

Reduction set

\( J_1, J_2, J_3, J_5 \)

\( v_1 \) \( J_1 \) \( v_2 \) \( J_2, J_3, J_5 \) \( v_3 \) \( J_4 \) \( v_4 \)

[0,0] [7,13] [10,19] [19,23]

No deadline miss

One single state
Partial-order reduction: how?

Before exploring a state:
- Form a candidate reduction set
- Derive response time bounds for the jobs in the reduction set
- Let the original schedule-abstraction graph explore all scheduling decisions
- Potential deadline miss
- No deadline miss
- Reduce the set to a single edge and form the new state

**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 \textbf{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)

---

Potential availability:
- **Potentially available from 7**
- **Certainly available from 13**

Initial candidate reduction set:

- Core: $J_1, J_2, J_3, J_4$
- Priority:
  - High: $J_1, J_2$
  - Low: $J_3, J_4$

**Low Priority**
- $J_1$: Available from 6.8, Executed from 22
- $J_2$: Available from 11.14, Executed from 20.23

**High Priority**
- $J_3$: Available from 11, Executed from 21
- $J_4$: Available from 21, Executed from 23
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 before or during 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.

---

**core**

- Potentially available
- Certainly available

**Priority**

- High
- Low

**Initial candidate reduction set $J^M$**

$J_1  J_2  J_3  J_4$
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 $J^M$.

Initially candidate reduction set $J_M$

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

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 $J^M$.

Initial candidate reduction set $J^M$

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.
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 \( J^M \).

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

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

See Lemma 5 in the paper
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)
Partial-order reduction for schedule-abstraction-based response-time analysis

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”

Max \{ $A^{\text{max}}$, $r_{3}^{\text{max}} - 1$\} + $B$

Potential availability ($A^{\text{min}}$)

Certainly availability ($A^{\text{max}}$)

Job under consideration

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 \( r_{\text{max}} \)

**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 \( r_{\text{max}} \) 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]
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?

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

<table>
<thead>
<tr>
<th>Preemption</th>
<th>Clock speed</th>
<th>#tasks/ runnables</th>
<th>#jobs per hyperperiod</th>
<th>Jitter</th>
<th>CPU time (s)</th>
<th>#states</th>
<th>Average #jobs per edge</th>
<th>Speedup</th>
<th>State-reduction ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Original</td>
<td>POR</td>
<td>Original</td>
<td>POR</td>
<td></td>
</tr>
<tr>
<td>Task-level</td>
<td>1.2 GHz</td>
<td>7</td>
<td>786</td>
<td>No</td>
<td>0.00538</td>
<td>0.00574</td>
<td>809</td>
<td>766</td>
<td>1.027</td>
</tr>
<tr>
<td></td>
<td>(U ≈ 0.18)</td>
<td></td>
<td></td>
<td>Yes</td>
<td>0.00951</td>
<td>0.00441</td>
<td>1248</td>
<td>586</td>
<td>1.344</td>
</tr>
<tr>
<td>Runnable-level</td>
<td>400 MHz</td>
<td>710</td>
<td>37,459</td>
<td>No</td>
<td>782.044</td>
<td>20.968</td>
<td>1,076,504</td>
<td>15,261</td>
<td>2.455</td>
</tr>
<tr>
<td></td>
<td>(U ≈ 0.53)</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Out of memory</td>
<td>N/A</td>
<td>351</td>
<td>107.026</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Preemption</th>
<th>Clock speed</th>
<th>#tasks/runnables</th>
<th>#jobs per hyperperiod</th>
<th>Jitter</th>
<th>CPU time (s)</th>
<th>#states</th>
<th>Average #jobs per edge</th>
<th>Speedup</th>
<th>State-reduction ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Original</td>
<td></td>
<td>Original</td>
<td>POR</td>
<td></td>
</tr>
<tr>
<td>Task-level</td>
<td>1.2 GHz</td>
<td>7</td>
<td>786</td>
<td>No</td>
<td>0.00538</td>
<td>809</td>
<td>1.027</td>
<td>0.937</td>
<td>5.315</td>
</tr>
<tr>
<td></td>
<td>(U ≈ 0.18)</td>
<td></td>
<td></td>
<td>Yes</td>
<td>0.00951</td>
<td>1248</td>
<td>1.344</td>
<td>2.157</td>
<td>53.045</td>
</tr>
<tr>
<td>Runnable-level</td>
<td>400 MHz (U ≈ 0.53)</td>
<td>710</td>
<td>37,459</td>
<td>No</td>
<td>782.044</td>
<td>1,076,504</td>
<td>2.455</td>
<td>37.298</td>
<td>98.582</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td>Out of memory</td>
<td>N/A</td>
<td>107.026</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

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

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

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

higher

Case study: automotive task sets

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

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

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

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

<table>
<thead>
<tr>
<th>Preemption</th>
<th>Clock speed</th>
<th>#tasks/runnables</th>
<th>#jobs per hyperperiod</th>
<th>Jitter</th>
<th>CPU time (s)</th>
<th>#states</th>
<th>Average #jobs per edge</th>
<th>State-reduction ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Original</td>
<td>POR</td>
<td>0.00538</td>
<td>766</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.00951</td>
<td>0.00441</td>
<td>1.027</td>
<td>53.045</td>
</tr>
<tr>
<td>Task-level</td>
<td>1.2 GHz (U ≈ 0.18)</td>
<td>7</td>
<td>786</td>
<td>Yes</td>
<td>0.00951</td>
<td>0.00441</td>
<td>1.344</td>
<td>98.582</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No</td>
<td>782.044</td>
<td>20.968</td>
<td>2.455</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td>Out of memory</td>
<td>13.996</td>
<td>37.298</td>
<td></td>
</tr>
<tr>
<td>Runnable-level</td>
<td>400 MHz (U ≈ 0.53)</td>
<td>710</td>
<td>37,459</td>
<td>No</td>
<td>1,076,504</td>
<td>15,261</td>
<td>107.026</td>
<td></td>
</tr>
</tbody>
</table>

State-reduction ratio = \[\frac{1 - \text{States explored by SAG + POR}}{\text{States explored by original SAG}}\]

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

States explored by SAG + POR

States explored by original SAG

Runtime of the original SAG

Runtime of SAG + POR

Partial-order reduction for schedule-abstraction-based response-time analysis
Scalability of partial-order reduction

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

61 out of 1400 task sets timed out

CPU time (s)

- 0.0
- 0.5
- 1.0
- 1.5

number of jobs

- $0 \times 10^6$
- $2 \times 10^4$
- $4 \times 10^4$
- $6 \times 10^6$

0 min
5 min
16 min
4 hours
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
- Scalability: millions of jobs instead of tens of thousands
- State-space reduction: 98% on average
- Over-approximation of the WCRT: 0.1% on average

Thank you ...

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

Partial-order reduction for schedule-abstraction-based response-time analysis