Reachability-Based Response-Time Analysis: Motivation, Challenges, and Open Problems

Mitra Nasri

In collaboration with
Geoffrey Nelissen and Björn Brandenburg

and the support of our PhD and Master students at TU/e and TUDelft
(Pourya Gohari, Sayra Ranjha, Suhail Nogd, Srinidhi Srinivasan, Joan Marce i Igual, ...)

Invited talk at RTSOPS

July 2022
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
Response-time analysis problem

This problem is NP-Complete for most job-level fixed-priority (JLFP) scheduling policies (even on a single-core platform).
Response-time analysis: where are we?

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

- Fast
- Pessimistic
- Hard to extend

\[ R_i^{(0)} = C_i + \sum_{j=1}^{i-1} C_j \]
\[ R_i^{(k)} = C_i + \sum_{j=1}^{i-1} \left( \frac{R_j^{(k-1)}}{T_j} \right) C_j \]

Longest blocking


Where has it taken us?

Experiment: limited-preemptive scheduling of parallel DAG tasks
Setup: 16 cores, 10 periodic DAG tasks

utilization

schedulability ratio

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8

[ISORC’17]
Response-time analysis: where are we?

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

- Accurate
- Easy to extend
- Not scalable

Setup: sequential non-preemptive periodic tasks scheduled by global fixed-priority scheduling policy (FP)
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 (a reachability-based analysis)
- Applicable to complex problems
- Highly accurate
- Scales

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, ... ]
Response-time analysis: where are we?

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

Schedule-abstraction graph (a reachability-based analysis)
- Applicable to complex problems
- Highly accurate
- Scales

Exact analyses in generic formal verification tools (e.g., UPPAAL)
- Accurate
- Easy to extend
- Not scalable

Experiment: limited-preemptive scheduling of parallel DAG tasks
Setup: 16 cores, 10 periodic DAG tasks

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
(a reachability-based analysis)
- Applicable to complex problems
- Highly accurate
- Scales

Setup: sequential non-preemptive periodic tasks, scheduled by global FP

Schedule-abstraction graph (on 8 cores)
Schedule-abstraction graph in a nutshell

- Workload model (timing features of the workload)
- Resource model (CPU)
- Scheduling policy (job-level fixed-priority policies)

Schedule-abstraction graph (a reachability-based response-time analysis framework)

- Merging
- Pruning
- Partial-order reduction

Response-time bounds

3000 times faster than generic verification tools (e.g., UPALL)

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

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

• Why response-time analysis is hard?

• The core idea (of our reachability-based response-time analysis)

• Current achievements

• Challenges, open problems, and next steps
Why the problem is hard?

One of the simplest forms of the problem:

**Response-time analysis problem**

*Given*
- a set of non-preemptive 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-preemptive 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**

<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>$J_1$</td>
<td>0 0</td>
<td>10</td>
<td>1 2</td>
<td>high</td>
</tr>
<tr>
<td>$J_2$</td>
<td>0 0</td>
<td>30</td>
<td>7 8</td>
<td>medium</td>
</tr>
<tr>
<td>$J_3$</td>
<td>0 15</td>
<td>30</td>
<td>3 13</td>
<td>low</td>
</tr>
<tr>
<td>$J_4$</td>
<td>10 10</td>
<td>20</td>
<td>1 2</td>
<td>high</td>
</tr>
</tbody>
</table>

**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 is generated by a fixed-priority scheduling policy on one core)

Q: Why can’t we “simulate” one schedule using a discrete-event simulator and see if there will be a deadline miss?

<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></td>
</tr>
<tr>
<td>$J_1$</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>high</td>
</tr>
<tr>
<td>$J_2$</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>medium</td>
</tr>
<tr>
<td>$J_3$</td>
<td>0</td>
<td>15</td>
<td>30</td>
<td>low</td>
</tr>
<tr>
<td>$J_4$</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>high</td>
</tr>
</tbody>
</table>

Earliest release time
Latest release time
BCET
WCET
Why the problem is hard?

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

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

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:
- $J_1$: 2, 10
- $J_2$: 10
- $J_3$: 0, 12, 21, 30
- $J_4$: 10

Example for path 2:
- $J_1$: 1
- $J_2$: 9
- $J_3$: 0, 10
- $J_4$: 24, 26

- 2 possible job orderings
- 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”
Agenda

• Why response-time analysis is hard?

• The core idea
  (of our reachability-based response-time analysis)

• Current achievements

• Challenges, open problems, and next steps
Response-time analysis using schedule-abstraction graphs

A path aggregates all schedules with the same job ordering

A path represents a set of similar schedules

Different paths have different job orders
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

$J_1: [4, 8]$ when it is dispatched after state $v$

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

Interpretation of an uncertainty interval:

- **Certainly not available**
- **Possibly available**
- **Certainly available**
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)

Expansion rules imply the scheduling policy

We will come back to this later in the presentation

Available jobs (at the state)

State $v_i$
\begin{align*}
\text{Core 1:} & \quad 10 \quad 15 \quad 20 \\
\text{Core 2:} & \quad 30
\end{align*}

Next states

J1 $\rightarrow$ High priority
J2 $\rightarrow$ Medium priority
J3 $\rightarrow$ Low priority

Release jitter

Available jobs

$J_1$

$J_2$

$J_3$

[ECRTS’2018]
Building the schedule-abstraction graph

**Building the graph** (a breadth-first method)

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)

- Initial state
- System is idle and no job has been scheduled
- Merging actions
- Repeat until every path includes all jobs
- All jobs have been scheduled
Why does our analysis scale?

Even though there are fewer job orderings than schedules, they are still **exponential** (e.g., under large release jitter or execution-time variations)

Without state-merging techniques, it is a **combinatorial problem**

⇒ The size of the graph grows exponentially

We **merge similar states** to curb the state-space explosion
Why does our analysis scale?

Our analysis for multicore platforms, uses the symmetry property to reduce the state space.
A closer look at the schedule-abstraction graph
Back to our example

System is idle and no job has been scheduled

**Single core, JLFP policy**

<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>J₁</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>high</td>
</tr>
<tr>
<td>J₂</td>
<td>0</td>
<td>30</td>
<td>7</td>
<td>medium</td>
</tr>
<tr>
<td>J₃</td>
<td>0, 15</td>
<td>30</td>
<td>3</td>
<td>low</td>
</tr>
<tr>
<td>J₄</td>
<td>10</td>
<td>20</td>
<td>1</td>
<td>high</td>
</tr>
</tbody>
</table>

Release jitter:

- J₁: [0, 0]
- J₂: [1, 2]
- J₃: [11, 22] = [10+1, 10+2]
- J₄: [12, 24]

Deadline miss for J₄:

**Uncertainty interval:**

- Possibly available: [a, b]
- Certainly not available: [a, b]
- Certainly available: [a, b]

Job Release time

Deadline

Execution time

Priority
Back to our example

<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>$J_1$</td>
<td>0 0</td>
<td>10</td>
<td>1 2</td>
<td>high</td>
</tr>
<tr>
<td>$J_2$</td>
<td>0 0</td>
<td>30</td>
<td>7 8</td>
<td>medium</td>
</tr>
<tr>
<td>$J_3$</td>
<td>0 15</td>
<td>30</td>
<td>3 13</td>
<td>low</td>
</tr>
<tr>
<td>$J_4$</td>
<td>10 10</td>
<td>20</td>
<td>1 2</td>
<td>high</td>
</tr>
</tbody>
</table>

If two paths have the same set of jobs, we can merge them.

Single core, JLFP policy

State merging may result in pessimism unless the resource-availability intervals of the two paths intersect.
How to use schedule-abstraction graphs to solve a new problem?

- Define the state abstraction
- Define the expansion rules
- Define merging rules

What is encoded by an edge? What is encoded by a state?
How to create new states?
How to identify similar states?

And then, prove soundness
“the expansion rules must cover all possible schedules of the job set”
Agenda

• Why response-time analysis is hard?

• The core idea
  (of our reachability-based response-time analysis)

• Current achievements

• Challenges, open problems, and next steps
Our journey

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

**Single core** [RTSS’17]

**Multicore** [ECRTS’18]

Independent jobs, global scheduling (bounds: close-to-tight)

**DATE’19**

Limited-preemptive tasks, global scheduling, tight bounds (UPPAAL)
Taste of results: sequential tasks (global scheduling)

Comparison with an **exact schedulability test** implemented in UPPAAL [DATE’19]

**Almost as accurate as the exact test**

**Much faster**

this work (8 cores)

Mitra Nasri – Reachability-based response-time analysis: motivation, challenges, and open problems
Taste of results: sequential tasks (global scheduling)

Setup: 10 non-preemptive periodic tasks, 4 cores, varying utilization

Our results

Improvement of more than 60 percentage point
Our journey

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

[RTSS’17] An end-to-end toolchain (from code to response-time analysis)

[ECRTS’18] Multicore

- Independent jobs, global scheduling (bounds: close-to-tight)

[DATE’19]

- Limited-preemptive tasks, global scheduling, tight bounds (UPPAAL)

[ECRTS’19] Parallel DAG tasks, global scheduling (bounds: not tight)
Taste of results: parallel DAG tasks (global scheduling)


Setup: 10 periodic DAG tasks
Our journey

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

[RTSS’17]

An end-to-end toolchain (from code to response-time analysis)

[RTSS’19]

Independent jobs, global scheduling (bounds: close-to-tight)

[ECRTS’18]

[DATE’19]

Jobs access shared data (FIFO and Priority-based spinlocks), global scheduling (bounds: not tight)

[RTSS’20]

[ECRTS’22]

Moldable Gang tasks
Our journey

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

**Single core**

- [RTSS'17]
- An end-to-end toolchain (from code to response-time analysis)
- [RTAS'22]
  - **Best paper award**
  - **Partial order reduction** (5 orders of magnitude faster)

**Multicore**

- [ECRTS'18]
  - Independent jobs, global scheduling (bounds: close-to-tight)
  - [DATE'19]
  - Limited-preemptive tasks, global scheduling (tight bounds (UPPAAL))
  - Partial order reduction (5 orders of magnitude faster)

- [ECRTS'19]
  - Parallel DAG tasks, global scheduling (bounds: not tight)
  - [RTSS'20]
  - Jobs access shared data (FIFO and Priority-based spinlocks), global scheduling (bounds: not tight)

- [ECRTS'22]
  - Moldable Gang tasks

**Distributed systems**

- [RTNS'22]
  - Analyzing data-age for multi-rate DAG tasks with data dependencies (bounds: not tight)
Partial-order reduction: the idea

In some cases, activities can interleave, and result in different end states

How did we reduce these unnecessary interleavings?

The idea of partial-order reduction (POR)

Do not explore interleavings that have no impact on the worst-case behavior
### Partial-order reduction for response-time analysis

<table>
<thead>
<tr>
<th>Job</th>
<th>Release time</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>20</td>
<td>(high)</td>
</tr>
<tr>
<td>$J_2$</td>
<td>6</td>
<td>1</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>$J_3$</td>
<td>7</td>
<td>2</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>$J_4$</td>
<td>18</td>
<td>3</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>$J_5$</td>
<td>9</td>
<td>5</td>
<td>25</td>
<td>(low)</td>
</tr>
</tbody>
</table>

**Exploring job schedules is irrelevant when**

- all those schedules lead to the **same system state**
- no job has a **deadline miss**

**Reduction set**

$\left\{ J_2, J_3, J_5 \right\}$

**Merged into single state**

**No deadline misses**
Partial-order reduction: how?

Before exploring a state:
- Form a candidate reduction set
- Derive response time bounds for the jobs in the reduction set

Potential deadline miss:
- Let the original schedule-abstraction graph explore all scheduling decisions

No deadline miss:
- Reduce the set to a single edge and form the new state

A significant reduction in the number of states
- States explored by SAG + POR
- States explored by original SAG

Higher

5 orders of magnitude faster
- Runtime of the original SAG
- Runtime of SAG + POR

Higher

Negligible impact on the WCRT estimate
- WCRT reported by SAG + POR
- WCRT reported by original SAG
Agenda

• Why response-time analysis is hard?

• The core idea (of our reachability-based response-time analysis)

• Current achievements

• Challenges, Open problems, and next steps
Open problems and next steps

**Improving scalability**

- Partial order reduction techniques (for multicore)
- Use new (and more efficient) abstractions
Open problems and next steps

Improving scalability

Extending the analysis to new problems (or problem domains)

Solve new problems:
- Preemptive systems
- Communication delays in networks
- End-to-end timing analysis of distributed systems and ROS-based applications
- Bus and memory access interference
- Sporadic arrivals
Open problems: analyzing sporadic tasks

**Hypothesis:** we can create an execution scenario (generated by the given scheduling policy) in which the job under analysis misses its deadline

- May hold? -> unschedulable
- Cannot hold? -> schedulable

How can we design a backward solution to automatically verify the hypothesis?
Open problems and next steps

- Improving scalability
- Extending the analysis to new problems (or problem domains)
- Automating the above extensions
A step towards automated response-time analysis

Our biggest next step

**Automating the rule generation** for schedule-abstraction graph for new (unforeseen) system models and scheduling policies (WiP: two PhD these)

**Requires**

- **A domain-specific language (DSL)** to describe system, resource, and workload model and the scheduling policy
- A **formal semantic** that connects the pieces of the DSL and their interactions
- An **automated way to generate rules** for the exploration phase of the schedule-abstraction graph
- An automated way to *generate a proof of soundness*
Open problems and next steps

- Improving scalability
- Extending the analysis to new problems (or problem domains)
- Automating the above extensions

Making it part of the design cycle (by adding robustness analysis, sensitivity analysis, etc.)

- Generate counter-examples
- Automatically configure the system such that there is no violation of timing properties
- Robustness analysis
Our work-in-progress on schedule-abstraction graph

- Partial-order reduction + **precedence constraints** (to be submitted)
- Support for **preemptive scheduling** on single-core platforms (to be submitted)

- **Partial-order-reduction** in multiprocessor platforms (WiP: MSc thesis)
- **Conditional DAG tasks** (uncertainty in program paths) (WiP: MSc thesis)
- Finding **counter examples** in case of deadline miss (WiP: MSc thesis)
- End-to-end latency analysis of **ROS applications** (WiP)

Our biggest next step

**Automating the rule generation** for schedule-abstraction graph for new (unforeseen) system models and scheduling policies (WiP: two PhD these)
In our RTAS’22 work, we made it **5 orders-of-magnitude faster** using partial-order-reduction.