Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling

Mitra Nasri
Geoffrey Nelissen
Björn Brandenburg

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS
TU Delft
CISTER
Research Centre in Real-Time & Embedded Computing Systems
Our work in a nutshell

We obtain the **worst-case and best-case response time**

**Workload model**
- Parallel DAG tasks (or job sets)

**Platform model**
- Multicore (identical cores)

**Scheduler model**
- Global job-level fixed-priority (JLFP)

**Execution model**
- Limited preemptive (fixed-preemption points)

**Task model**

**Arrival model**

**Execution model**

- Preemption points at segment boarders

- Bounded uncertainty

- Deadline (hard or soft)

- Worst case (WCET)

- Best case (BCET)

- bounded uncertainty
Our work in a nutshell

We obtain the **worst-case** and **best-case response time**

**Workload model**
- Parallel DAG tasks (or job sets)

**Platform model**
- Multicore (identical cores)

**Scheduler model**
- Global job-level fixed-priority (JLFP)

**Execution model**
- Limited preemptive (fixed-preemption points)

**Job model**
- Bounded non-deterministic arrivals
- Not sporadic tasks

Examples:
- Transactions
- Multi-frame tasks
- Periodic DAG tasks

This job model supports bounded non-deterministic arrivals, but not sporadic tasks (unbounded non-deterministic arrivals)
State of the art

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

- Fast
- Pessimistic
- Hard to extend

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

Formulations of lower-priority tasks. A response-time analysis of a DAG-based task-set with a limited-preemptive priority scheduler is computed by iterating the following equation until a fixed point is reached, starting with:

\[
R_k = \text{len}(G_k) + \frac{1}{m} \left( \text{vol}(G_k) - \text{len}(G_k) \right) \\
R_k \leftarrow \text{len}(G_k) + \frac{1}{m} \left( \text{vol}(G_k) - \text{len}(G_k) + I_k^{bp} + I_k^{lp} \right)
\]
State of the art

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

- Fast
- Pessimistic
- Hard to extend

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

- Accurate
- Easy to extend
- Not scalable
State of the art

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

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

This line of work

**Response-time analysis using schedule abstraction**
- Applicable to complex problems
- Easy to extend
- Highly accurate
- Relatively fast

Industrial use cases are typically large, complex, and require accurate analysis
State of the art: comparison
Experiment on sequential periodic tasks

### Accuracy

<table>
<thead>
<tr>
<th>number of tasks</th>
<th>schedulability ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>0.75</td>
</tr>
<tr>
<td>9</td>
<td>0.50</td>
</tr>
<tr>
<td>12</td>
<td>0.25</td>
</tr>
<tr>
<td>15</td>
<td>0.00</td>
</tr>
</tbody>
</table>

- **exact test (timeout)**
- **this paper**
- **exact test (UPPAAL)**

#### 4 cores, 30% utilization

**Almost as accurate as the exact test**

### Runtime

<table>
<thead>
<tr>
<th>number of tasks</th>
<th>runtime (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>500</td>
</tr>
<tr>
<td>9</td>
<td>1,000</td>
</tr>
<tr>
<td>12</td>
<td>1,500</td>
</tr>
<tr>
<td>15</td>
<td>2,000</td>
</tr>
<tr>
<td>18</td>
<td>2,500</td>
</tr>
<tr>
<td>21</td>
<td>3,000</td>
</tr>
</tbody>
</table>

- **4 cores**
- **2 cores**
- **8 cores**
- **1 core**

**Much faster**

Experiment on sequential periodic tasks (global FP scheduling)
State of the art: comparison

Effectiveness (for parallel DAG tasks)

Much less pessimistic than the closed-form analysis
State of the art: schedule-abstraction-based analyses

**[RTSS’17]**

<table>
<thead>
<tr>
<th>Uniprocessor</th>
<th>Independent non-preemptive jobs/tasks</th>
<th>Work-conserving and non-work-conserving job-level fixed-priority scheduling (JLFP)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exact</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**[ECRTS’18]**

<table>
<thead>
<tr>
<th>Multiprocessor</th>
<th>Independent non-preemptive jobs/tasks</th>
<th>Global work-conserving job-level fixed-priority scheduling (JLFP)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sufficient</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**[this work]**

<table>
<thead>
<tr>
<th>Multiprocessor</th>
<th>Non-preemptive jobs/DAG tasks with precedence constraints</th>
<th>Global work-conserving job-level fixed-priority scheduling (JLFP)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sufficient</strong></td>
<td></td>
<td>A new system abstraction (more scalable)</td>
</tr>
</tbody>
</table>

[RTSS’17] M. Nasri and B. Brandenburg, "An Exact and Sustainable Analysis of Non-Preemptive Scheduling".
Agenda

• Schedule-abstraction-based analysis

• Supporting precedence constraints
  • Challenges
  • A new abstraction

• Evaluation

• Conclusion and future work
Response-time analysis using schedule abstraction
Highlights

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

Observation

There are fewer permissible job orderings than schedules.

Solution

Use job-ordering abstraction to analyze schedulability by building a graph that represents all possible schedules.
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

\( J_1: [4, 8] \)

Earliest and latest finish time of \( J_1 \) when it is dispatched after state \( \nu \)
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:
- Possibly available
- Certainly available
- Certainly not available

A system state

Core 1:
- Start: 10
- End: 30

Core 2:
- Start: 15
- End: 20

Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling
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 = \( \min \{ \text{completion times of the job} \} = 2 \)

Worst-case response time = \( \max \{ \text{completion times of the job} \} = 15 \)
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
Building the schedule-abstraction graph

**Expanding a vertex:**
(reasoning on uncertainty intervals)

Expansion rules imply the scheduling policy

![Diagram of schedule-abstraction graph]

**State** $v_i$

- **Core 1:**
  - Task 1: 10
  - Task 2: 30
- **Core 2:**
  - Task 1: 15
  - Task 2: 20

**Next states**

- **$J_1$**
  - Available jobs: 35, 40
  - Task 1 duration: 17
  - Task 2 duration: 30
  - Priority: High
- **$J_2$**
  - Available jobs: 8, 25
  - Task 1 duration: 17
  - Task 2 duration: 25
  - Priority: Medium
- **$J_3$**
  - Available jobs: 35, 40
  - Task 1 duration: 17
  - Task 2 duration: 40
  - Priority: Low
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"
• Schedule-abstraction-based analysis

• Supporting precedence constraints
  • Challenges
  • A new abstraction

• Evaluation

• Conclusion and future work
Handling precedence constraints

An example schedule:

- **Core 1:**
  - $J_1$ with $BCET = 2$ and $WCET = 12$
  - Release time: 3, Completion time: 15

- **Core 2:**
  - $J_2$ with $CET = 12$
  - Release time: 7, Completion time: 10

- **Core 3:**
  - $J_3$ with $CET = 12$
  - Release time: 2, Completion time: 9

- **Core 4:**
  - $J_4$ cannot become ready before time 7

Is that enough?

- $C_1 \in [2, 12]$ (tikz node $J_1$)
- $C_2 \in [6, 9]$ (tikz node $J_2$)
- $C_3 \in [2, 5]$ (tikz node $J_3$)

The latest time at which all predecessors have been completed.
Challenge 1: modeling precedence constraint as release jitter may cover impossible scenarios (→ pessimism)

An example schedule:

Core 1: $J_1$

Core 2: $J_2$

Is there a scenario at which two cores are busy at any time in the interval $[1, 3]$?

No! because $J_2$ can start its execution only if $J_1$ has finished
Challenge 1’s solution: keep track of running jobs in a state

Maintain a set $S$ of certainly running jobs and their completion time intervals.

When scheduling $J_i$: Remove all its predecessors from $S$.

Update availability of cores considering the removal of $J_i$’s predecessors.

Running jobs:
- Core 1: idle
- Core 2: idle

Initial state:
- Core 1: idle
- Core 2: idle
- Running jobs: \{\}

Schedule $J_1$:
- Core 1: 1
- Core 2: idle
- Running jobs: \{\}$J_1: [1, 3]$

Schedule $J_2$:
- Core 1: 3
- Core 2: idle
- Running jobs: \{\}$J_2: [3, 5]$

Core states:
- Core 1: idle
- Core 2: idle

Task $J_1$ and $J_2$ completion times:
- $J_1$: [1, 5]
- $J_2$: [3, 5]
Challenge 2: Updating certainly running jobs after the merge phase

\[ S = \{ J_1: [2, 5], J_2: [7, 10] \} \]

\[ S = \{ J_1: [2, 8] \} \]

\[ S = \{ J_1: [4, 8], J_3: [3, 9] \} \]

\[ v_p \text{ merged with } v_q \]
Challenge 3: improving the scalability (with a new state abstraction)

Prior work [ECRTS’18]:

Assume that these jobs can be scheduled on either of the cores

Symmetry increases the cost of “expansion phase”

Note: these vertices will likely “merge” during the merge phase anyway
Our new state abstraction

[ECRTS’18]

Core 1

\[ EFT_1 \]

Core 2

\[ EFT_2, LFT_2 \]

Core 3

\[ EFT_3, LFT_3 \]

This work

1 core

\[ A_1^{\text{min}}, A_1^{\text{max}} \]

2 cores

\[ A_2^{\text{min}}, A_2^{\text{max}} \]

3 cores

\[ A_3^{\text{min}}, A_3^{\text{max}} \]

How does it help?

When a new job is dispatched, it only affects the first core availability interval

because there is no need to expand all combination of jobs and cores!

\[ EFT_i \]: earliest finish time of the \( i^{th} \) core
\[ LFT_i \]: latest finish time of the \( i^{th} \) core
\[ A_i^{\text{min}} \]: earliest availability time of \( i \) cores
\[ A_i^{\text{max}} \]: latest availability time of \( i \) cores
Evaluating the effect of the new abstraction

Non-preemptive periodic tasks, 4 cores, utilization = 70%

---

Graph showing the 95th percentile runtime for different numbers of tasks, comparing [ECRTS'18] and This work. The graph indicates a significant improvement, approximately 12x faster in the 95th percentile runtime for 30 tasks compared to [ECRTS'18].
Agenda

- Schedule-abstraction-based analysis
- Supporting precedence constraints
  - Challenges
  - A new abstraction
- Evaluations
- Conclusion and future work

Evaluation
Experiment setup

Experiment platform
• Multi-threaded C++ program. We parallelized the breadth-first exploration of the schedule-abstraction graph using Intel’s open-source Thread Building Blocks (TBB) library.

• A cluster of machines each equipped with 256 GiB RAM and Intel Xeon E5-2667 v2 processors clocked at 3.3 GHz.

• We report the CPU time of all of the threads together as the runtime of the analysis.

DAG tasks:
• Periods in [500, 100000]
• Utilization of a task: uUniFast

• Series-parallel DAGs with nested fork-joins generated with the method from [Cassini 2018, Serrano 2017, Melani 2015, Peng 2014]
  • Maximum nodes in a DAG: 50
  • Maximum length of the critical path: 10
  • Maximum nested branches: 3
DAG tasks: varying cores (m) and tasks (n)

U=50%

Scalability experiment

More cores than tasks: higher parallelism

More tasks than cores: each task has smaller utilization

Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling
DAG tasks: varying cores (m) and tasks (n)

U=50%
Conclusions and future directions
Conclusion

Response-time analysis using schedule abstraction

+ New abstraction

+ Expansion rules to support precedence constraints

Results: achieving high accuracy (similar to UPPAAL) while being able to scale to practically relevant system sizes $(n \leq 20, m \leq 64)$
Questions

This work
- Multiprocessor
- Parallel DAG tasks
- Better state abstraction
- Global job-level fixed-priority scheduling (JLFP)

Future work
- Near future
  - Preemptive execution
  - Partial-order reduction
  - Heterogeneous platforms
  - Self-suspending tasks
- In a few years
  - Sporadic tasks
  - Gang scheduling
  - Shared resources
  - Co-running tasks
- Eventually
  - Dynamic schedulers
  - Combine the framework with timing analysis tools

The framework is open source. You can find that on the authors’ page.

Thank you