An Exact and Sustainable Analysis of Non-Preemptive Scheduling

Mitra Nasri*        Björn B. Brandenburg

Max Planck Institute for Software Systems (MPI-SWS)
Germany

RTSS, December 2017
A wide class of non-work conserving scheduling algorithms

Non-preemptive jobs (or periodic tasks)

Exact schedulability analysis

Exact best-case and worst-case response time

Fixed-job priority scheduling algorithms

Uniprocessor

Release jitter

Execution time variation

Hard or soft timing constraints

Our work in a nutshell

“An exact and sustainable schedulability analysis for non-preemptive scheduling”
Why non-preemptive scheduling?

Examples
- GPU device
- Hardware accelerators
- CAN bus

Inevitable
(where preemption is not supported by the platform/network)

Improves timing predictability

Improves QoS

Simplifies system design

Low overhead

- Control systems are sensitive to I/O delay and preemptions
- Simpler resource management policies
- Grants exclusive resource access

- A more accurate estimation of worst-case execution-time (WCET)
- More predictable cache

- Reduces context switches
- Avoids intra-task cache-related preemption delays (CRPD)
Most of NPS policies are **not sustainable** (w.r.t. execution time variation, release jitter, etc.)

- Schedulability analyses for **sporadic tasks**
  - [Jeffay91, Tindel94, Davis07]

- **Pessimistic for periodic tasks**

- **Not applicable to arbitrary job sets**

Existing schedulability analyses based on model checking, timed automata, abstraction refinements, etc.
- [Sun97, Baker07, Guan07, Bonifaci10, Burmyakov15, Stigge15]

- **Not very scalable**

- Existing analyses are not efficient

- Existing analyses are not enough

- Simulation-based schedulability tests cannot be used

- No solution yet

Many non-work-conserving scheduling algorithms do NOT have a schedulability analysis yet
An efficient, exact, general schedulability analysis that includes a wide class of scheduling algorithms and task models.
Main idea:

Searching all possible schedules efficiently and accurately

- Constructing the search graph
- Evaluation
- Conclusion
Basic scenario: no runtime variation in the workload

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_3)</td>
<td>30</td>
<td>13</td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>30</td>
<td>8</td>
</tr>
<tr>
<td>(\tau_1)</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Non-preemptive fixed-priority scheduling

One schedule

One job ordering

Values are integer.

Scheduling algorithm: Non-preemptive fixed-priority (NP-FP)

A schedule is an assignment of execution intervals to the jobs.

Both existing tests for sporadic tasks reject this task set [Jeffay91, Davis07]
### Scenario: execution time variation and release jitter

<table>
<thead>
<tr>
<th>Task \ Period</th>
<th>Execution time</th>
<th>Release jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_3 ) \ (30)</td>
<td>[3, 13]</td>
<td>15</td>
</tr>
<tr>
<td>( \tau_2 ) \ (30)</td>
<td>[7, 8]</td>
<td>0</td>
</tr>
<tr>
<td>( \tau_1 ) \ (10)</td>
<td>[1, 2]</td>
<td>0</td>
</tr>
</tbody>
</table>

Values are integer.

Scheduling algorithm: NP-FP

A schedule is an assignment of execution intervals to the tasks.
Is there a way to use job-ordering abstraction to analyze schedulability?

For an **exact** analysis, we need to consider all **possible execution scenarios**

**Observation**

There are fewer permissible **job orderings** than **schedules**

Due to scheduling anomalies

**Research question**

**Is there a way to use job-ordering abstraction to analyze schedulability?**

**How to abstract**

schedules in a graph of job orderings?

**How to efficiently**

find all job orderings?

**How to identify**

timing violations in the resulting graph?
Abstracting schedules in a graph of job orderings

Requirement
Knowing when a job misses its deadline

Solution
Encode the earliest and latest finish time of a job

Verification of schedulability
Check if the latest finish time is not larger than the deadline

Each path shows a job ordering

Deadline of $J_{1,2}$ is at time 20

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Min</th>
<th>Max</th>
<th>Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_3$</td>
<td>10</td>
<td>[3, 13]</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>30</td>
<td>[7, 8]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>30</td>
<td>[1, 2]</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Agenda

- Main idea: Searching all possible execution scenarios efficiently and accurately
- Constructing the search graph
- Evaluation
- Conclusion
Constructing the search graph

```
Start

Sort the jobs according to their priorities (scheduling policy)

Create the first vertex \( v_1 \) with interval \([0, 0]\)

Is there a path that can be expanded?

Yes

Select the shortest path \( P \)

Find eligible jobs

For each eligible job, find the earliest and latest finish time and add them to \( P \)

Merge any two paths that share the same set of jobs

Report deadline misses

end

No

Job set

Fixed-job-priority scheduling algorithm

Initialization

Merge two paths if they have the same set of jobs and their final intervals intersect

Grow

Breadth-first search

The graph grows more slowly
```
Growing the graph

Start

Sort the jobs according to their priorities (scheduling policy)

Create the first vertex $v_1$ with interval $[0, 0]$

Is there a path that can be expanded?

Yes

Select the shortest path $P$

Find eligible jobs

For each eligible job, find the earliest and latest finish time and add them to $P$

Merge any two paths that share the same set of jobs

Report deadline misses

No

An eligible job for path $P$ is a job that can be scheduled after $P$ in at least one execution scenario

Path $P$

$v_1 \rightarrow \ldots \rightarrow J_3 \rightarrow v_i \rightarrow J_4 \rightarrow v_j \rightarrow v_{j+1}$

$e_i = \text{the earliest finish time of path } P$

$l_i = \text{the latest finish time of path } P$
Requirements of an exact analysis

“Eligibility conditions” are necessary and sufficient

The “final interval” of each is exact:
For any time \( t \) in the interval, there must be an execution scenario that ends at \( t \)

Final intervals remain “exact” after merging process

In our work, we have proved these properties for
- Fixed-job-priority scheduling algorithms
- Tasks with release jitter and execution time variation
- Hard and soft timing constraints
- Work-conserving and non-work-conserving scheduling algorithms
How to apply the analysis to a new system or algorithm?

1. Define eligibility conditions
2. Define how to obtain the final intervals
3. Prove the aforementioned properties
Agenda

- Main idea: Searching all possible execution scenarios efficiently and accurately
- Constructing the search graph

Evaluation

Conclusion
Main questions

- Is our analysis **effective**?
  - Does it actually improve the accuracy of schedulability analysis?
  - What is our **achievement** for **non-work-conserving** scheduling policies?

- Is our analysis **efficient**?
  - How fast is the analysis?
Evaluation setup

**Automotive benchmark task sets [Kramer15]**

- Variable parameter: utilization
- Generate runnables according to [Kramer15] until the given utilization is reached
- Pack a random number of runnables together to build a task
- Up to 30 tasks per task set

**Synthetic task sets**

- Variable parameter: maximum number of jobs in a hyperperiod
- Periods are from [1, 1000] ms with log-uniform distribution
- Up to 50% runtime variation in the execution time
- 10 tasks per task set

**No jitter**

**Small jitter** (up to 100 microseconds)

**Large jitter** (up to 20% of the period)

To evaluate the effectiveness in a realistic setup and different utilization values

To evaluate the efficiency when there are a large number of jobs

---


**Note:** only task sets that pass the necessary schedulability condition of non-preemptive scheduling were considered.
Task sets in this experiment have up to 35 tasks and 3500 jobs
How effective is our schedulability analysis?

- NP-FP classic test
- This paper: NP-EDF
- This paper: NP-FP

Still, many task sets are not schedulable

About 40% more schedulable task sets are found

Are these task sets not schedulable by any algorithm?

Automotive benchmark, no jitter
How effective is our schedulability analysis?

- NP-FP classic test
- This paper: NP-EDF
- This paper: NP-FP
- This paper: Precautious-RM
- This paper: CW-EDF+

Automotive benchmark, no jitter

Non-work-conserving policies
How efficient is our schedulability analysis?

### No jitter

- Red: This paper: NP-EDF
- Black dashed: This paper: NP-FP
- Blue plus: This paper: Precautious-RM
- Orange: This paper: CW-EDF+

<table>
<thead>
<tr>
<th>Utilization</th>
<th>CPU time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>0.9</td>
<td>0.9</td>
</tr>
</tbody>
</table>

### Large jitter

- Orange: This paper: CW-EDF+

<table>
<thead>
<tr>
<th>Utilization</th>
<th>CPU time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>0.9</td>
<td>0.9</td>
</tr>
</tbody>
</table>

About 1.5 hours automotive benchmark
How efficient is our schedulability analysis?

- this paper: NP-EDF
- this paper: NP-FP
- this paper: P-RM
- this paper: CW-EDF+

CPU time (sec)

maximum number of jobs (per hyperperiod)

Synthetic tasks
Small jitter

About 30 minutes
Main idea: Searching all possible execution scenarios efficiently and accurately

Constructing the search graph

Evaluation

Conclusion
Conclusion

Goal
An efficient, exact, and general schedulability analysis for a wide class of scheduling algorithms

Solution
Constructing a precise abstraction of all possible schedules

Method
Building a schedule-abstraction graph based on job ordering

Key idea
An efficient merge technique to defer the state-space explosion
Future directions

- Preemptive and limited preemptive scheduling
- Multiprocessor systems
- Global and semi-Partitioned scheduling
- Shared resources
- Precedence Constraints
- Parallelizing the analysis framework to make it even faster

Our analysis
Our analysis

- Non-preemptive job set (or periodic tasks)
- Hard or soft timing constraints
- Uniprocessor
- Release jitter
- Execution time variation
- Exact best-case and worst-case response time
- A wide class of non-work conserving scheduling algorithms
- Fixed-job priority scheduling algorithms

Source code available at

Thank you