Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions

Daniel Casini*, Alessandro Biondi*, Geoffrey Nelissen†, and Giorgio Buttazzo*

* ReTiS Lab, Scuola Superiore Sant’Anna, Pisa, Italy
† CISTER, ISEP, Polytechnic Institute of Porto, Portugal
Overview

Partitioned Fixed-Priority Scheduling of Parallel Tasks

Each task is represented by a Direct Acyclic Graph, and is characterized by

i. a minimum inter-arrival time $T_i$

ii. a constrained deadline $D_i \leq T_i$

iii. a fixed priority $\pi_i$
Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions

• Each task is represented by a **Direct Acyclic Graph**, and is characterized by
  
  i. a minimum inter-arrival time $T_i$
  
  ii. a **constrained deadline** $D_i \leq T_i$
  
  iii. a **fixed priority** $\pi_i$
**Overview**

Partitioned

Fixed-Priority Scheduling of Parallel Tasks Without Preemptions

- Each node is statically assigned to a core
- Nodes of the same task can be allocated to different cores
Overview

Partitioned Fixed-Priority Scheduling of Parallel Tasks

Without Preemptions

Non-preemptive blocking

As soon a node starts executing, it cannot be preempted
Why non-preemptive scheduling?

Predictable management of local memories

e.g., nodes can pre-load data from scratchpads before start executing

Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures

Daniel Casini, Alessandro Biondi, Geoffrey Nelissen and Giorgio Buttazzo

This morning @ RTSS
Why non-preemptive scheduling?

- Predictable management of local memories
- Reduces context-switch overhead
- Simplifies WCET Analysis
- Use of HW accelerators and GPUs
- Can be a good choice for executing deep neural networks
Why non-preemptive scheduling?

- We profiled a deep neural network executed by Tensorflow on an 8-core Intel i7 machine @ 3.5GHz.

More than 34000 nodes where only about 1.2% of them have execution times larger than 100 microseconds.
Overview of the analysis framework

Parallel Tasks

Node-to-core Mapping

Core 1
Core 2
Core 3
Core 4

Self-suspending Tasks

Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks

Response-time analysis for parallel tasks

Schedulable? (yes/no)
Overview of the analysis framework

**Part I**

- **Response-time analysis for parallel tasks**
  - **Schedulable? (yes/no)**

**Part II**

- **Self-suspending Tasks**
  - Uniprocessor analysis for SS tasks

**Part III**

- **Node-to-core Mapping**
  - Core 1
  - Core 2
  - Core 3
  - Core 4

- **Parallel Tasks**
Overview of the analysis framework

Part I

Response-time analysis for parallel tasks

Schedulable? (yes/no)

Part II

Self-suspending Tasks

Uniprocessor analysis for SS tasks

Part III

Node-to-core Mapping

Parallel Tasks

Core 1
Core 2
Core 3
Core 4

Core 1
Core 2
Core 3
Core 4

Part II

Self-suspending Tasks

Uniprocessor analysis for SS tasks

Part I

Response-time analysis for parallel tasks

Schedulable? (yes/no)
Part I: Response-time analysis for parallel tasks
• Each core ‘perceives’ the execution of a parallel task as an interleaved sequence of execution and suspension regions.

Suspension regions correspond to execution regions on a different core.
• **Paths can be mapped to a self-suspending tasks**

![Diagram showing paths and regions]

- Paths can be mapped to a self-suspending tasks.
**Intuition**

- **Paths can be mapped to a self-suspending tasks**

The length of each execution region directly maps to the WCET of a node in the graph.
The length of each suspension region depends on the response time of nodes allocated to different cores.

Complex inter-core dependencies can arise.
• Recursive algorithm to unfold response-time dependencies:

\[ v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \]

\[ S_1^{SS1} = R^{SS2} \]

\[ S_1^{SS2} = R^{SS3} \]

\[ S_1^{SS3} = R^{SS4} \]
Parallel tasks without preemptions

• We extended this approach to work under non-preemptive scheduling

Need for a fine-grained analysis for non-preemptive self-suspending tasks

Our next contribution
Part II: Analysis for non-preemptive self-suspending tasks
Overview of the analysis framework

Parallel Tasks

Node-to-core Mapping

Core 1
Core 2
Core 3
Core 4

Self-suspending Tasks

Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks

Response-time analysis for parallel tasks

Schedulable? (yes/no)
Overview of the analysis for SS-tasks

Two different approaches:

1. **Holistic analysis**
   - Computes the RT of a *whole* self-suspending task
   \[ C_i = \sum_{\text{all segments}} C_{i,j} \]
   \[ S_i = \sum_{\text{all segments}} S_{i,j} \]
   - **Analytically dominates** state-of-the-art analysis (Dong et al. 2018)

2. **Segment-based analysis**
   - Computes the RT of *individual* segments
   \[ C_i = \sum_{\text{all segments}} C_{i,j} \]
   \[ S_i = \sum_{\text{all segments}} S_{i,j} \]
Overview of the analysis for SS-tasks

Two different approaches:

1. Holistic analysis
   - Computes the RT of a whole self-suspending task
   
   \[ C_i = \sum C_{ij} \]

2. Segment-based analysis
   - Computes the RT of individual segments

Hybrid model: Take the minimum of the two bounds

per-segment response time bounds
Analysis for SS-tasks

**Interference** due to higher-priority segments

Response-time analysis

Non-preemptive **blocking** due to lower-priority segments
Interference from higher-priority tasks is accounted by means of the following worst-case scenario*:

- Interference from higher-priority tasks is accounted by means of the following worst-case scenario*:

  - The response time bound can be initially approximated to the task’s deadline and iteratively refined.

    - **Holistic** and **segmented** analyses are combined during the iterative refinement.

Fine-grained accounting of blocking

- With a multiset approach

\[ \Delta \]

\[ \tau_{HP} \]

\[ \tau_{LP} \]

Two segmented SS-tasks contending for the same CPU

Contains the WCET of all the lower-priority segments that may block the task under analysis in a window of length \( \Delta \)

multiset
Fine-grained accounting of blocking

- With a **multiset** approach

\[ \tau_{HP} \]

\[ \tau_{LP} \]

\[ \Delta \]

Two segmented SS-tasks contending for the same CPU

Contains the WCET of all the lower-priority segments that may block the task under analysis in a window of length \( \Delta \)

**multiset**
Non-preemptive self-suspending tasks

Now we have our analysis!
Non-preemptive self-suspending tasks

Now we have our analysis!
Non-preemptive self-suspending tasks

Now we have our analysis!

The RT of a parallel task can be derived from the maximum RT of all its paths

\[ R = \max(R', R'', R''') \]
Overview of the analysis framework

Parallel Tasks

Node-to-core Mapping

Core 1
Core 2
Core 3
Core 4

Self-suspending Tasks

Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks
Uniprocessor analysis for SS tasks

Response-time analysis for parallel tasks

Schedulable? (yes/no)
Overview of the analysis framework

Parallel Tasks

Core 1

Core 2

Core 3

Core 4

Node-to-core Mapping

Self-suspending Tasks

How to partition nodes to cores?

Uniprocessor analysis for SS tasks

Uniprocessor analysis for SS tasks

Uniprocessor analysis for SS tasks

Uniprocessor analysis for SS tasks

Response-time analysis for parallel tasks

Schedulable? (yes/no)
**Partitioning (meta-)algorithm**

**IDEA:** Analyzing schedulability *incrementally*, adding one node at a time, and perform *schedulability analysis* on a subgraph.

**Inputs:**
1. Strategy for ordering *tasks*
2. Strategy for ordering *cores*

**Output:**
1. Node partitioning

**Example:**

- **Task under analysis**
- **Task under analysis (during partitioning)**

- **Core 1**
- **Core 2**
- **Core 3**
- **Core 4**
**Partitioning (meta-)algorithm**

**IDEA:** Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

**Inputs:**
1. Strategy for ordering tasks
2. Strategy for ordering cores

**Output:**
1. Node partitioning

**Example:**
- Task under analysis
- Task under analysis (during partitioning)
- Core 1
- Core 2
- Core 3
- Core 4
Partitioning (meta-)algorithm

**IDEA:** Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

**Inputs:**
1. Strategy for ordering tasks
2. Strategy for ordering cores

**Output:**
1. Node partitioning

**Example:**
- Task under analysis
- Task under analysis (during partitioning)
IDEA: Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph

Inputs:
1. Strategy for ordering tasks
2. Strategy for ordering cores

Output:
1. Node partitioning

Example:
Run analysis for parallel tasks
IDEA: Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

Inputs:
1. Strategy for ordering tasks
2. Strategy for ordering cores

Output:
1. Node partitioning

Example:
Run analysis for parallel tasks

Schedulable!
IDEA: Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph

Inputs:
1. Strategy for ordering tasks
2. Strategy for ordering cores

Output:
1. Node partitioning

Example:

Task under analysis

Task under analysis (during partitioning)
Partitioning (meta-)algorithm

**IDEA:** Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

**Inputs:**
1. Strategy for ordering tasks
2. Strategy for ordering cores

**Outputs:**
1. Node partitioning

**Example:**

Task under analysis

Task under analysis (during partitioning)
**Partitioning (meta-)algorithm**

**IDEA:** Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

**Inputs:**
1. Strategy for ordering *tasks*
2. Strategy for ordering *cores*

**Output:**
1. Node partitioning

**Example:**

Run analysis for parallel tasks

Task under analysis

Task under analysis (during partitioning)
**Partitioning (meta-)algorithm**

**IDEA:** Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

**Inputs:**
1. Strategy for ordering tasks
2. Strategy for ordering cores

**Output:**
1. Node partitioning

**Example:**
- Task under analysis
- Task under analysis (during partitioning)
- Run analysis for parallel tasks
- Unschedulable!

**Core 1**
**Core 2**
**Core 4**
IDEA: Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

Inputs:
1. Strategy for ordering tasks
2. Strategy for ordering cores

Output:
1. Node partitioning

Example:

Task under analysis

Task under analysis (during partitioning)
Partitioning (meta-)algorithm

IDEA: Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph

Inputs:
1. Strategy for ordering tasks
2. Strategy for ordering cores

Output:
1. Node partitioning

Example:

- Task under analysis
- Task under analysis (during partitioning)

Run analysis for parallel tasks
Partitioning (meta-)algorithm

IDEA: Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

Inputs:
1. Strategy for ordering tasks
2. Strategy for ordering cores

Output:
1. Node partitioning

Example:
Run analysis for parallel tasks
Schedulable!
### IDEA: Partitioning (meta-)algorithm

Analyzing schedulability **incrementally**, adding one node at a time, and perform schedulability analysis on a subgraph.

#### Inputs:
1. Strategy for ordering **tasks**
2. Strategy for ordering **cores**

#### Output:
1. Node **partitioning**

#### Example:

- **Task under analysis**

- **Task under analysis (during partitioning)**
IDEA: Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph

Inputs:
1. Strategy for ordering tasks
2. Strategy for ordering cores

Output:
1. Node partitioning

Example:

Task under analysis

Task under analysis (during partitioning)
**Partitioning (meta-)algorithm**

**IDEA:** Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph.

**Inputs:**
1. Strategy for ordering **tasks**
2. Strategy for ordering **cores**

**Output:**
1. **Node partitioning**

**Example:**

**Task under analysis**

**Task under analysis (during partitioning)**

**Core 1**

**Core 2**

**Core 3**

**Core 4**
Partitioning (meta-)algorithm

IDEA: Analyzing schedulability incrementally, adding one node at a time, and perform schedulability analysis on a subgraph

Inputs:
1. Strategy for ordering tasks
2. Strategy for ordering cores

Output:
1. Node partitioning

Example:

Task under analysis

Task under analysis (during partitioning)

Partitioning completed!
Experimental Results
Experimental Study

- Experimental study based on synthetic workload
  - We compared against the only previous work targeting non-preemptive scheduling of parallel tasks, which targets global scheduling (Serrano et al. 2017)
  - Same DAG generator used in [Serrano et al. 2017]

- WCETs randomly generated in $(0, 100]$ with uniform distribution
- Tasks utilizations obtained with U-Unifast
- Tasks periods computed as $T_i = U_i \sum_{nodes} c_{i,j}$
Experimental Results

12 tasks, 16 processors

Schedulability ratio

Utilization

The higher the better
Experimental Results

12 tasks, 16 processors

Schedulability ratio

Utilization

Increasing task-set utilization
Experimental Results

12 tasks, 16 processors

Utilization

Schedulability ratio

Improvement up to 100 percentage points over [Serrano et al. 2017]
Experimental Results

Our experimental study revealed a similar trend varying the number of tasks and processor, e.g.,

10 tasks, 8 processors

Utilization vs. Schedulability ratio
Conclusions

1. **Methodology** for analyzing non-preemptive parallel tasks as a set of self-suspending tasks

2. **Analysis** for non-preemptive self-suspending tasks which analytically dominates the only previous result

3. **Partitioning algorithm** to allocate nodes to the available processors

4. **Experimental study** to assess the improvement in terms of schedulability – up to 100 p.p. w.r.t. the only existing previous work for global scheduling
Future Work

Deeper investigation of partitioning strategies

Improvement in the analysis precision

Integration of communication delays in the analysis

Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures

Daniel Casini, Alessandro Biondi, Geoffrey Nelissen and Giorgio Buttazzo

This morning @ RTSS
Thank you!

Daniel Casini
daniel.casini@sssup.it