Response-Time Analysis for Task Chains in Communicating Threads (RTAS’16)

Johannes Schlatow, Rolf Ernst

April 14th, 2016
Introduction

- growing variety and complexity (e.g. automotive domain)
- object-oriented and component-based design for reusability and separation (e.g. AUTOSAR)

→ interfaces with procedure call semantics (same core) (e.g. microkernel-based systems)
Motivational example

Two ADAS functions implemented by multiple software components:

- **Parking assistant** (P), trajectory calculation (T), object recognition (O₁)
- **Lane detection** (L), object recognition & object masking (O₂), steering (S)

How can we verify latency requirements of P and L?
Modelling communicating threads for timing analysis

Threads
- sequence of instructions and communication
- scheduled by the OS (here: static priority)

→ precedence constraints (dependencies) between thread segments

Tasks
- activated by preceding task (or external stimulus)
- communicate at completion
- activations can queue up
- execute on the thread‘s priority

Threads
- sequence of instructions and communication
- scheduled by the OS (here: static priority)

→ precedence constraints (dependencies) between thread segments

Tasks
- activated by preceding task (or external stimulus)
- communicate at completion
- activations can queue up
- execute on the thread's priority
Observations

1. Task graph obfuscates procedure call (synchronous) semantics.
   - Caller is blocked until the callee returns
   → non-overlapping execution of task chains
   → predecessors cannot interfere with dependent tasks (pessimistic results)

Example:

- **Synchronous semantics** (non-overlapping execution):
  - Task chain latencies

- **Asynchronous semantics** (overlapping execution):
  - Task chain latencies

Diagram showing task chain latencies and priority levels.
Observations

1. Task graph obfuscates procedure call (synchronous) semantics.
   - caller is blocked until the callee returns
   → non-overlapping execution of task chains
   → predecessors cannot interfere with dependent tasks (pessimistic results)

2. Task chains have non-monotonic priorities.
   - in contrast to: descending priority assignment
   → task-chain latency = response time of last task
Observations

1. Task graph obfuscates procedure call (synchronous) semantics.
   - caller is blocked until the callee returns
   - non-overlapping execution of task chains
   - predecessors cannot interfere with dependent tasks (pessimistic results)

2. Task chains have non-monotonic priorities.
   - in contrast to: descending priority assignment
   - task-chain latency = response time of last task

Wanted:
Worst-case latency analysis for task chains on the same resource that...
- considers the procedure call semantics of the thread communication
- can deal with non-monotonic, i.e. arbitrary priority assignments.
Outline

- Motivation
- Analysis approach and system model
- Response-time analysis for synchronous task chains
- Application to asynchronous task chains
- Related work
- Experimental evaluation
- Conclusion
Analysis approach – preliminary work

Analysis flow of Compositional Performance Analysis (CPA)

- **event model interface** $\eta^{+/-}(\Delta t)$: max/min number of activations between any time window $\Delta t$
- **local scheduling analysis** based on busy-window technique:Calculates amount of time a resource is busy processing $q$ events of task $i$.
  - e.g.:
    \[ B_i(q) = q \cdot C_i^+ + \sum_{j \in I_i} \eta_j^+(B_i(q)) \cdot C_j^+ \]

- **event model propagation**: Derives new event models based on local scheduling analysis results.
- **repeated until convergence**
- **path latency**: sum of WCRTs
**Analysis approach – modification**

**Problem:** Interference accounted multiple times within a task chain.
- Can be limited dependent on the semantics of the task chain.

**Idea:** busy-window analysis for entire task chains

→ q-event task-chain busy window

→ improvement of local scheduling analysis

→ applied but not limited to CPA
System Model

Assumptions

- static-priority preemptive (SPP) scheduling on processing resource
- task chains do not cross resource boundaries
- tasks within the chain have exactly one incoming and outgoing edge
- same communication semantics for entire resource (easily extensible)
- arbitrary priorities

Terminology

- a task chain \( i \) consists of a sequence of tasks \( (\tau_{i1}, \tau_{i2}, \ldots, \tau_{in_i}) \)
- synchronous task chain = non-overlapping execution
- asynchronous task chain = overlapping execution possible
- best-case/worst-case execution time for each job of \( \tau_{ik} \): \( \frac{C_{ik^-}}{C_{ik^+}} \)
Response time analysis for synchronous task chains

Intra-chain interference
- no overlapping execution

Inter-chain interference
- stalling and **deferred** activations:

```
priority

a0
a2
b0
b1
a1
```

```
chain under analysis

b0 → b1
a0 → a1 → a2
```

April 14th, 2016 | Johannes Schlatow, Rolf Ernst | Response-Time Analysis for Task Chains in Communicating Threads | Slide 12
Response time analysis for synchronous task chains

Intra-chain interference
- no overlapping execution

Inter-chain interference
- stalling and deferred activations:

Diagram showing two chains of tasks with overlaid priority levels and a busy window including deferred activation.
Task-chain busy window for synchronous task chains

Q-event task-chain busy window:

- self interference bounded by \( q \)
- considers every (non-deferred) task on a higher priority than any task in the chain: \( I_{ij} \)
- single-time blocking limited to the critical deferred segment \( S_{ij} \)

busy-window for task chain \( i \):

\[
B_i(q) = q \sum_k C_{ik}^+ + \sum_{j \neq i} \left( \sum_{j \in I_{ij}} \eta_j^+ (B_i(q)) \cdot C_{jk}^+ + \sum_{k \in S_{ij}} C_{jk}^+ \right)
\]

inter-chain interference

normal interference (bounded by \( \eta_{a}^+ \))

self interference (bounded by \( q \))

critical deferred segment

mutual-exclusive deferred segments

chain \( i \)

chain \( j \)
Task-chain busy window for synchronous task chains

Q-event task-chain busy window:
- self interference bounded by \( q \)
- considers every (non-deferred) task on a higher priority than any task in the chain: \( I_{ij} \)
- single-time blocking limited to the critical deferred segment \( S_{ij} \)

busy-window for task chain \( i \):

\[
B_i(q) = q \sum_k C_{ik}^+ + \\
\sum_{j \neq i} \left( \sum_{j \in I_{ij}} \eta_{jk}^+ (B_i(q)) \cdot C_{jk}^+ + \sum_{k \in S_{ij}} C_{jk}^+ \right) \\
\]

Can this be applied to asynchronous chains?
Application to asynchronous task chains

Q-event task-chain busy window for asynchronous task chains:

- additional self-interference
- deferred tasks $D_{ij} = \text{tasks dependent on a stalled task}$

$\rightarrow$ single-time blockers

**Q-event task-chain busy window equation:**

$$
B_i(q) = \eta_i^+(B_i(q)) \sum_k C_{ik}^+ + \sum_{j \neq i} \left( \sum_{j \in I_{ij}} \eta_j^+(B_i(q)) \cdot C_{jk}^+ + \sum_{k \in D_{ij}} C_{jk}^+ \right)
$$

- **self interference (bounded by $\eta_b^+$)**
- **deferred tasks (bounded by 1, proof in the paper)**
- **normal interference (bounded by $\eta_a^+$)**

**Diagram:**

- Graphical representation of task chains with nodes labeled M, H, L, indicating the progression of tasks through different stages.
Outline

- Motivation
- Analysis approach and system model
- Response-time analysis for synchronous task chains
- Application to asynchronous task chains
- Related work
- Experimental evaluation
- Conclusion
Related work

**Context-aware analysis extensions** (distributed systems)
- offset-based analyses [Palencia et al. 1999, Redell 2003, Henia et al. 2006]
- pay bursts only once [Schliecker et al. 2009]
- limiting event streams [Kollmann et al. 2010, 2011]

**Refinement of task models**
- classification and schedulability analysis [Stigge 2014]

→ no exploitation of (synchronous) communication semantics in chains on a single resource
Experimental evaluation

Implementation

- extension module for pyCPA
- requires small modification of pyCPA core (limit propagation)

Experiments

- synthetic experiments
  - conventional pyCPA (sum of tasks’ WCRTs)
  - task-chain busy window
- automotive use case (park assist + lane detection)
  - conventional pyCPA (sum of tasks’ WCRTs)
  - task-chain busy window
  - MAST (offset-based analysis with precedence relations)

pyCPA: http://bitbucket.org/pycpa
MAST: http://mast.unican.es/
Synthetic experiments – Setup

Comparison of conventional pyCPA with our extension for task chains.

- task set of six tasks with fixed WCET/BCET
- three different compositions into two chains (a & b)

utilisation: $U_{3:3} = 0.97 \mid U_{4:2} = 0.82 \mid U_{5:1} = 0.78$

- distinct task priorities
- ran analysis for all possible priority permutations in each composition
- compared resulting WCRTs of both task chains
Synthetic experiments – Synchronous

relative latency improvement:

\[
\text{task chain WCRT} = \frac{\sum_i WCRT_i}{\text{relative latency}}
\]

our results

conventional pyCPA

median improvement:
3:3) a: 0.18 | b: 0.19
4:2) a: 0.13 | b: 0.29
5:1) a: 0.13 | b: 0.6
Synthetic experiments – Asynchronous

- Smaller improvement due to self-interference

Median improvement:
- 3:3) a: 0.35 | b: 0.29
- 4:2) a: 0.17 | b: 0.33
- 5:1) a: 0.13 | b: 0.6

Relative latency task chain a vs. relative latency task chain b
Automotive use case

Parking assistant and lane detection (introductory example):

Task chain P
- period 200ms, jitter 5ms, core execution time 70ms

Task chain L
- period 100ms, jitter 5ms, core execution time 50ms

Objective: Find a feasible thread priority assignment under given latency constraint for both task chains (150ms).

→ analyse 5040 priority assignments
Automotive use case – Results summary

Conventional CPA:
- analysed 5040 priority assignments in about 8h (single core desktop)
- no convergence for all but 6 cases
- latency results between 4949 and 8613ms (P), 1017 and 2322ms (L)
  → deemed not feasible

MAST:
- analysis took 34 seconds, results for all 5040 priority assignments
- 11 assignments feasible (below the required maximum latency)

Task-chain busy window:
- analysis took 22 seconds, converged for all 5040 priority assignments
- 2880 assignments feasible (below the required maximum latency)
Automotive use case – Detailed latency results

The diagram illustrates the latency results for synchronous (sync), MAST, and asynchronous (async) task chains in a communicating threads context. The x-axis represents the latency task chain L, while the y-axis shows the latency task chain P. The data points are color-coded to differentiate between sync, MAST, and async tasks. The red line at 150 on the y-axis indicates a critical threshold for latency.
Summary & Conclusion

- Task chains resulting from **communicating threads** imply certain semantics.
- Improved **local scheduling analysis** for SPP-scheduled task chains.
- Improved **coverage** (# analysable systems).
- Much tighter (and realistic!) **WCRT results**.
- Reduced analysis **run-time** (from hours to seconds).
- Enables (in-field) **design-space exploration**.
- Enhances **applicability** of response-time analysis for existing software implementations (e.g. RTE, 3rd-party software stacks, libraries).

Thank you for your attention. Questions?
References