Self-aware scheduling for mixed-criticality component-based systems

Johannes Schlatow, Mischa Möstl and Rolf Ernst

RTAS 2019
Let’s decrypt the title...

Component-based systems

**Mixed-criticality** in component-based systems = **temporal isolation**
- we use **budget-based scheduling** (e.g. sporadic server scheduling)
  - guarantees service and limits interference
  - good fit for **component-based systems** (different vendors, blackbox components)
- **How much** budget does a component need?
  - **WCET not known** for blackbox components
  - budgetting for worst-case leads to **overreservation** (impractical)

**Self-aware scheduling = temporal reflection**
- continuous monitoring
- trace-based approach (less than worst-case)
Let’s decrypt the title…

Component-based systems

Overview

- specify/model application budgets
- efficiently enforce budgets by the scheduler
- continuously validate whether both, scheduler and application model, are correct

Self-aware scheduling

- continuous monitoring
- trace-based approach (less than worst-case)

Mixed-criticality systems

- we use budget-based scheduling (e.g. sporadic server scheduling)
- guarantees service and limits interference
- good fit for component-based systems (different vendors, blackbox components)

How much budget does a component need?

- WCET not known for blackbox components
- budgetting for worst-case leads to overreservation (impractical)
How to characterise application workload?

Components communicate using RPCs (call/return) or signals

- **client calls server** and waits for return
- **client signals** receiver every third activation

Observations

- server executes on **client’s scheduling context**
  - common in microkernels (donation)
- receiver executes on its **own scheduling context**
- varying execution times
Can we make use of (existing) sporadic server scheduling?

**Sporadic server scheduling**
- budget is **constant** within replenishment period $T_R$
- sporadic server is considered as **periodic task** in response-time analysis (RTA)

**What is the (best) $T_R$ in our example?**
- service (budget) is guaranteed
- **but**: RTA is pessimistic

→ sporadic server does not exactly match our workload
→ even worse as **implementations often use a fixed $T_R$**
Event-based replenishment

Looks like a workload arrival function (WAF)
- Apply shaping according to WAFs?
- Implementation must track when budget was consumed → time-based replenishment
- even sporadic server implementations are tricky [1]
- preemptions lead to replenishment fragmentation

Idea: avoid implementation complexity
- cumulative Execution Time model [2]:
  \[ ET_i(n) = \text{upper bound on the cumulative execution time of } n \text{ consecutive jobs of } \tau_i \]
- event-based replenishment

---

[1] Stanovich et al., “Defects of the POSIX sporadic server and how to correct them”, in RTAS 2010
Implementing event-based replenishment

**Replenish budget on job arrival according to predefined ET(n)**

- implemented in Genode OS Framework (default: deferrable server scheduling)
- **memory overhead**: storing ET(1..L) with maximum L=10 \(\rightarrow\) 161 Bytes / Thread
- **time overhead**: budget calculation on job arrival \(\rightarrow\) 336 cycles* for L=10

**Caveats:**

- job arrival rate must be bounded (**IRQ shaping** required)
- if a component is underbudgeted, it may never finish
  - use idle time \(\rightarrow\) **background scheduling**
  - notify components if budget expired \(\rightarrow\) **timeout exceptions**

*on ARM Cortex-A9
Implementing event-based replenishment

Replenish budget on job arrival according to predefined ET(n)
- implement on job arrival
- memory overhead: storing ET(1..L) with maximum L=10 → 161 Bytes / Thread
- time overhead: budget calculation on job arrival → 336 cycles* for L=10

Caveats:
- job arrival rate must be bounded (IRQ shaping required)
- if a component is underbudgeted, it may never finish
- use idle time → background scheduling
- notify components if budget expired → timeout exceptions

Can we monitor the ET(n) in order to...

1. Extract the budget required by a particular component.
2. Limit the interference that a budgeted component causes on other components.

*on ARM Cortex-A9
Architecture Overview

- multiple cores may be monitored by a **single monitoring core**
- **trace event** is written for every scheduling event (cycle counter as timestamp, 28 Bytes /event)
- **monitor** reads trace buffer **periodically** and performs checks (e.g. response-time analysis)
- budget is **(re)configured** on demand
Extraction of ET(n) curves

Monitoring

- trace buffer contains **trace events from scheduler**
- get **execution time trace** from stream of trace events
- calculate **ET(n)** from execution time trace

→ enables adaptation/validation
Considering scheduling overheads

If we grant a budget ET(n) to a component, what is...

- the **service guaranteed** to the component (lower bound)
  - *guaranteed service* $\geq$ *granted budget*
- the **maximum interference** on other components (upper bound)
  - *maximum interference* $\leq$ *granted budget* + *overheads*

What are the (scheduling) implementation overheads?

- How exact is the budget accounted?
- Scheduling is not a zero-time operation.
- How can we **model scheduling overheads**?
Modelling scheduling overheads and budget accounting

- kernel uses one-shot timer for time accounting
- evaluated and set to remaining budget on every kernel call:

![Diagram showing preemption and expiration]

Observations

- kernel execution time must be included in the granted budget
- timer is not strictly monotonic → **effective budget** > **granted budget**
- max interference = granted budget + \#preemptions \times P_{err} + \#expiration \times E_{err}
Estimation of overheads

How to determine $P_{err}$ and $E_{err}$?

- Overheads can be measured or determined by WCET analysis.
- May be very pessimistic (worst case rarely occurs).
- Instead: use continuous monitoring to determine and adapt to the typical case.

Overhead monitoring

- Overrun = \#preemptions $\times P_{err}$ + \#expirations $\times E_{err}$
- Many samples, two unknown variables ($P_{err}$ and $E_{err}$).
- Goal: find upper bound that minimises error for all samples.
- Offline method: linear programming.
- Online method: least squares approximation.

Details in the paper.
Evaluation

Can we deal with wrongly estimated execution budgets?
- e.g. lower-priority blocking (shared server)
- extract and adapt to increased budget requirements

Can we estimate scheduling overheads at runtime?
- tested with differently challenging workloads
Summary

Event-based replenishment

- lightweight implementation
- approximation of real-time workload similar to workload arrival curves
- reduces the number of replenishments/preemptions
- should be used with timeout exceptions to let application know about its budget expiration

Monitoring

- continuous validation/adaptation of application model (ET(n))
- continuous validation/adaptation of scheduler model (overheads)
- enables temporal reflection → self-aware scheduling

Thank you. It’s time for questions or comments.