Memory-Processor Co-Scheduling in Fixed Priority Systems

Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Giorgio Buttazzo
What will we schedule?

- **Hundreds** of cores
- **One** main memory

Memory bandwidth is becoming the scarce resource!
How to «schedule» memory accesses?

- **Very different** from scheduling tasks on CPUs!
  - Memory accesses are sparse
  - Much smaller granularity

- **Solution:** *increase* granularity
  - Memory accesses are concentrated at the beginning of the execution
  - Prefetching execution models
Prefetching execution model

- Related works:
  - Prefetch-based execution models
    - Automotive, avionics
    - Double buffering in HPC
  - PREM framework [Pellizzoni et al.]
- Tasks split into different phases
  1. **Memory phase**: task context loaded into local memory
Prefetching execution model

- Related works:
  - **Prefetch**-based execution models
    - Automotive, avionics
    - Double buffering in HPC
  - **PREM** framework [Pellizzoni et al.]

- Tasks split into different phases
  1. **Memory phase**: task context loaded into local memory
  2. **Execution phase**: task executes with no contention for shared resources

- Main advantages
  - Easier timing analysis
  - Hide memory latencies
  - Exploit write/read burst features of DMA engines
Our contribution

- Current schedulability analyses are **pessimistic** or based on **heuristics**
- We want to follow a **systematic, analytic** approach

Simplest case: **single core** and **single memory channel**

Identify a **critical instant** scenario

Derive an **exact** response-time analysis
System model

- **M/C task model**
  - $n$ sporadic real-time tasks $\tau_1, \ldots, \tau_n$
  - $\tau_i = (M_i, C_i)$
    - $M_i$: worst-case memory-access time
    - $C_i$: worst-case execution time
- Constrained relative deadlines $D_i \leq T_i$

- M- and C- phases of different instances may overlap
  - They access different resources
  - Local memory may be partitioned

- **Fixed-priority preemptive** scheduler
  - Same priority on the processor and for accessing memory
  - Preemption overheads neglected
Analyses

Real-time **distributed computing** problem
- «Chains» of tasks
- Different processing nodes
- End-to-end deadlines

**Flow-shop** scheduling problem
- Context of production scheduling
- Makespan minimization
- Optimal poly-time solution if both stages have a single resource (Johnson’s algorithm)
Negative results - I

- **EDF is not optimal**
Negative results - II

- Synchronous periodic arrival pattern is **not** the critical instant

\[ \tau_1 \]
\[ \tau_2 \]
\[ \tau_1 \] postponed by 1

Synchronous periodic

Higher priority

23rd International Conference on Real-Time Networks and Systems
Lille (France), 4th November 2015
Negative results - II

• Synchronous periodic arrival pattern is not the critical instant

Higher priority

What is the critical instant scenario?
Each higher-priority task has **at most one** dual-interfering job!
Critical instant scenario

Start from the synchronous periodic arrival pattern ...

\[ R_k^M \text{ is maximized} \quad \checkmark \quad R_k^C \text{ is not} \quad \times \]
Critical instant scenario

... shift right all interfering tasks until they have a dual-interfering job whose M/C point is aligned with that of $\tau_k$
... shift right all interfering tasks until they have a dual-interfering job whose M/C point is aligned with that of $\tau_k$

$R^M_k$ does not change

$R^C_k$ can only increase
Critical instant scenario

... shift right all interfering tasks until they have a dual-interfering job whose M/C point is aligned with that of $\tau_k$

$R_k^M$ does not change

$R_k^C$ further increases
Exact schedulability test

- The response-time of both phases is maximized under critical instant

\[ R_k = R_k^M + R_k^C \]

- **Memory-phase** response-time

\[ R_k^M = \sum_{i \leq k} \left\lfloor \frac{R_k^M}{T_i} \right\rfloor M_i \]  

Same as in the classical case (no precedence constraints)

- **Computation-phase** response-time

\[ R_k^C = C_k + \sum_{i < k} \left\lfloor \frac{R_k^C + R_i^M}{T_i} \right\rfloor C_i \]  

Must account for “release jitter” of memory phase
Experimental setting

- Synthetic M/C task generation [1]
  - Worst-case execution time $C_k$ selected in [10, 1000]
  - Worst-case memory access time $M_k$ computed as $\lfloor f_{mc} C_k \rfloor$
    - $f_{mc} = M_k / C_k$ is the memory-to-computation ratio
    - Task utilization $u_k = u_k^M + u_k^C$ generated by UUniFast
    - Period computed as $T_k = \left\lfloor \frac{M_k + C_k}{u_k^M + u_k^C} \right\rfloor$
    - Relative deadline selected in $[M_k + C_k, T_k]$
    - **Deadline Monotonic (DM)** priority ordering

- Results in terms of **number of schedulable task-sets** and **weighted schedulability**

Experimental results

Varying utilization, $f_{mc} = 0.5, n = 8$

Varying M to C ratio, $n = 8$

[RTA-MC] Our exact response-time analysis for M/C task-sets


[RTA] Response-time analysis for sequential task-sets, with execution time $E_k = M_k + C_k$
Experimental results

Varying number of tasks, $f_{mc} = 0.1$

Constrained deadlines

Implicit deadlines

Better pipelining
Conclusions

- Identified a **critical instant scenario** for M/C task-sets scheduled under **fixed priority** in a single-core/single-memory setting.

- **Exact response-time analysis** for such task-sets:
  - Same complexity as classic RTA for sequential task-sets
  - Leverages pipelining of memory and execution phases

- Our results confirm the **great potential** of prefetching execution models:
  - Effectively hide memory latency
  - Significant schedulability improvement over sequential models

A lot of work still remains to be done ...
Future work

• Compiler-based tools to produce M/C-compliant code
• Priority assignment problem
• Schedulability analysis in more general settings
  – Multi-processor, single-memory
  – Multi-processor, multi-memory
• Generalization to multi-phase tasks
• Integrating preemption overheads
  – Account for memory penalties
  – Limited preemption framework
Thank you!

Marko Bertogna
marko.bertogna@unimore.it