Analysis and Implementation of Global Preemptive Fixed-Priority Scheduling with Dynamic Cache Allocation

Meng Xu   Linh Thi Xuan Phan   Hyon-Young Choi   Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Trend: Multicore proliferation

• Example: Automotive systems
  – Tesla Model S adopts Tegra 3 quad-core processor with a 1MB shared cache

• Example: Unmanned vehicles
  – DJI Matrice 100 uses ARM A15 quad-core with a 4MB shared cache

• Benefits: Reduce cost, increase performance, etc.
**Problem: Shared cache interference**

- Concurrent tasks may access the same region of the cache → cache misses
  - **BAD**: Increase WCETs of tasks
    - Experiment on PARSEC benchmark shows 41% increase in WCET
  - **WORSE**: Difficult to bound overhead
    - In the worst case, need to assume every access is a miss!
Common solution: Static cache alloc.

- **Pros**: Simple

- **Cons**: Low cache resource utilization
  - Unused cache area of one task cannot be reused by another

- **Cons**: Not always feasible
  - e.g., when the whole task set does not fit into the cache
Approach: Dynamic cache allocation

- Dynamically assign disjoint cache areas to jobs when they are released and resumed
- **Pros:** Better cache resource utilization
- **Pros:** Tasks can have larger cache areas $\rightarrow$ smaller WCETs
- **Challenge:** Difficult to implement and analyze
  - A task may be preempted by a higher-priority task via both CPU and cache resources
  - Result: More complex schedulability analysis and overhead scenarios
Contributions

• The gFPca algorithm
  – A new global preemptive fixed-priority cache-aware scheduling algorithm with dynamic cache allocation

• Implementation of gFPca in LITMUS\textsuperscript{RT}

• Overhead-aware schedulability analysis for gFPca
  – A new overhead-accounting approach for gFPca

• Extensive evaluation
  – Empirical experiments on an ARM platform
  – Numerical schedulability evaluation
Outline

• Introduction
• $gFPca$ algorithm and its implementation
• Empirical evaluation
• Overhead-aware schedulability analysis
• Numerical evaluation
System model

- Multi-core platform w/ a cache shared among cores
  - The shared cache is divided into multiple partitions

- Cache-aware sporadic tasks
  - Task: (Period, Deadline, \( \text{WCET}, \# \text{cache partitions} \))
  - A task cannot execute unless it has reserved the number of cache partitions it requires
gFPca Algorithm: Key ideas

- **Globally** schedule tasks based on their (fixed) priorities
- A higher-priority task can **preempt** one or more lower-priority tasks to acquire CPU or cache resources
- Lower-priority tasks can execute when pending higher-priority tasks are unable to execute
- **Dynamically** allocate cache partitions to a job when it is released or resumed
gFPca: Example

- Four tasks: T1 > T2 > T3 > T4
- Number of cache partitions: T1 = 14, T2 = 4, T3 = 4, T4 = 2

T1 need to preempt both T2 and T3 to get sufficient cache
2 partitions left → only T4 can execute
Implementation on Freescale IMX6Q

- **PL310 cache controller**: used to divide the cache into 16 cache partitions
- OS can configure the cache lockdown register to assign specific cache partitions to a core

![Diagram showing running tasks and cores with lockdown registers and shared cache]
Implementation on Freescale IMX6Q

- When a job is preempted: configure the core’s lockdown register to unlock its cache partitions
- When a job is scheduled to run on a core: configure the core’s lockdown register to lock the partitions for the job

![Diagram showing the implementation on Freescale IMX6Q](image-url)
Outline

• Introduction

• gFPca algorithm and its implementation

• Empirical evaluation

• Overhead-aware schedulability analysis

• Numerical evaluation
Implementation overhead: Methodology

- Randomly generated tasks that each read an array of size uniformly distributed in [64KB, 1MB]

- Randomly generated periodic task sets
  - Task set size: from 50 to 450 tasks, with a step of 50
  - 10 task sets per task set size (i.e., 90 task sets in total)

- Schedulers considered
  - gFPca: our proposed cache-aware scheduler
  - nFPca: cache-aware global non-preemptive FP [Guan, EMSOFT’09]
  - gEDF: cache-agnostic global EDF scheduling

- For each scheduler
  - Traced each task set for 30 seconds
  - Measured several types of overhead: release overhead, scheduling overhead, context switch overhead, IPI delay
Scheduling overhead

**Average overhead under cache-read workloads**

gFPca has **reasonable** avg. scheduling overhead (≤ 20μs)
Context switch overhead: Cache-read

Average overhead under cache-read workloads

Cache-aware schedulers have higher context switch overhead (Context switches involve cache flushing, due to **HW limitation**).
Context switch overhead: CPU-bound

Average overhead under CPU-bound workloads

Cache-aware schedulers have similar overhead as cache-agnostic schedulers
Scheduling performance of gFPca

• Implemented four schedulers in LITMUSRT
  – **gFPca**: our proposed cache-aware algorithm
  – **gFP**: global FP scheduling
  – **nFPca**: cache-aware global non-preemptive FP [Guan EMSOFT'09]
  – **pFPca**: partitioned scheduling with static core-level cache allocation

• Workload: real-time programs
  – T1: (period=5000ms, deadline =500ms, WCET=430ms, #CPs=16)
  – T2 – T5: (period=5000ms, deadline =1550ms, WCET=800, #CPs=4)
  – Priority: T1 > T2 > T3 > T4 > T5

The workload is only schedulable under gFPca scheduler
Outline

• Introduction
• gFPca algorithm and its implementation
• Empirical evaluation
  • Overhead-aware schedulability analysis
• Numerical evaluation
Overhead under gFPca

- Our analysis accounts for all 5 types of overhead
- This talk: **Cache overhead accounting**
Cache-related overhead

- When a task accesses a cache partition that is in the cache, it experiences a cache hit.
Cache-related overhead

• When a task is preempted, its cache content may be evicted out by other running tasks

• When the task resumes, it may need to reload the evicted cache content and experience cache misses
Challenge

- Three tasks: T1 > T2 > T3
- Number of cache partitions: T1 = 4, T2 = 14, T3 = 4
- T2 cannot execute due to insufficient cache resources

T1 finishes $\rightarrow$ T2 is scheduled $\rightarrow$ T3 is preempted

How to tightly account for the overhead?
Insight: Analyze source events that cause overhead

- Five types of task events
  - Task-migration event does not cause overhead in a shared cache
  - Source events: release and finish events

- Idea #1: Analyze the overhead caused by each source event to a task

- Idea #2: Eliminate source events that do not cause overhead using necessary conditions of task preemptions

- Combine the results to derive the total overhead a task experiences
## Overhead caused by a task-release event

<table>
<thead>
<tr>
<th>Case</th>
<th>Task i does not need to preempt task k</th>
<th>Task i can preempt task k via cache resource</th>
<th>Task i can preempt task k via CPU resource only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>None</td>
<td>Reloading time to reload cache partitions evicted by high-priority and low-priority tasks</td>
<td>Reloading time to reload cache partitions evicted by low-priority tasks</td>
</tr>
</tbody>
</table>

- **Case 1:** Task i does not need to preempt task k
  - Overhead: None
  - Task i can preempt task k via cache resource: Reloading time to reload cache partitions evicted by high-priority and low-priority tasks
  - Task i can preempt task k via CPU resource only: Reloading time to reload cache partitions evicted by low-priority tasks

### Diagram

- **P0** and **P1** are target tasks.
- **CP0**, **CP1**, **CP2**, and **CP3** are cache partitions.
- **Red** bars represent high-priority tasks.
- **Blue** bars represent low-priority tasks.

### Diagram Notes

- In the case where task i does not need to preempt task k, there is no overlap in resource usage.
- In the case where task i can preempt task k via cache resource, cache partitions evicted by both high- and low-priority tasks need to be reloaded.
- In the case where task i can preempt task k via CPU resource only, cache partitions evicted by low-priority tasks need to be reloaded.
## Overhead caused by a task-release event

<table>
<thead>
<tr>
<th>Case</th>
<th>Task i <strong>does not need</strong> to preempt task k</th>
<th>Task i can <strong>preempt</strong> task k via cache resource</th>
<th>Task i can <strong>preempt</strong> task k via CPU resource only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>None</td>
<td>Reloading time to reload cache partitions evicted by high-priority and low-priority tasks</td>
<td>Reloading time to reload cache partitions evicted by low-priority tasks</td>
</tr>
</tbody>
</table>

The overhead caused by one task-release event of a task i to a task k:

\[
\Delta_{i,k}^r = PRT \cdot |\phi_{i,k}^r| \quad \text{where} \quad \phi_{i,k}^r = \begin{cases} 
0 & \text{(Case 1)} \\
UCP_k \cap \left( ECP_i \cup \bigcup_{k<l, A_l<A_k} ECP_l \right) & \text{(Case 2)} \\
UCP_k \cup \left( \bigcup_{k<l, A_l<A_k} ECP_l \right) & \text{(Case 3)}
\end{cases}
\]
Overhead caused by a job of task $i$

- A job has one task-release event and one task-finish event
- The overhead caused by one task-release event of task $i$ to task $k$ is $\Delta_{i,k}^r$
- The overhead caused by one task-finish event of task $i$ to task $k$ is $\Delta_{i,k}^f$

The overhead caused by one job of task $i$ to task $k$ is

$$
\delta_{i,k}^r \overset{\text{def}}{=} \max \{\Delta_{i,k}^r, \Delta_{i,k}^f\}, \quad
\Delta_{i,k}^r + \Delta_{i,k}^f - \min_{j<k, j \neq i} \{\Delta_{j,k}^r, \Delta_{j,k}^f\}
$$

Either task-release event or task-finish event causes overhead

Both task-release and task-finish events of the same job causes overhead

(There must exist a task-release or task-finish event of another task $j$ that cannot cause overhead to task $k$)
Overhead experienced by task k

- The overhead experienced by a task $k$ is bounded by

$$\delta^k = \sum_{i=1}^{k-1} \delta^k_i \cdot \left[ \frac{d_k}{p_i} \right] + \Delta_{i,k}^f + \Delta_{i,k}^r$$

- Inflated WCET of task $k = \text{its WCET} + \delta^k$

A task set is schedulable under gFPca in the presence of overhead if its inflated task set is deemed schedulable by gFPca overhead-free analysis.
Outline

• Introduction
• gFPca algorithm and its implementation
• Empirical evaluation
• Overhead-aware schedulability analysis
• **Numerical evaluation**
Methodology

- Randomly generated sporadic tasks
  - Period: uniformly distributed in [550, 650]ms
  - Utilization: uniformly distributed in [0.5, 0.9]
  - Number of partitions: uniformly distributed in [1, 12]

- Randomly generated sporadic task sets
  - Task set utilization: from 0.1 to 4, with a step of 0.1
  - Number of task sets per task set utilization: 100

- Platform: 4 cores, 16 cache partitions

- Schedulers considered: gFPca, nFPca, gFP

- For each scheduler
  - Measured overhead on Freescale IMX6Q board
  - Analyzed the schedulability of each task set
  - Computed the fraction of schedulable task sets per task set util.
Schedulability analysis results

Both cache-aware gFP outperform cache-agnostic gFP

gFPca outperforms nFPca in general
Conclusion

• We introduced gFPca, a new cache-aware scheduling algorithm with **dynamic** cache allocation
  – Provides cache isolation and better cache resource utilization

• An implementation of gFPca on an ARM Cortex A9 board
  – For comparison, we also implemented gFP, nFPca and pFPca

• Overhead-aware schedulability analysis
  – Tightly accounts for the overhead based on the source events that cause overhead and necessary conditions of preemption events

• Our evaluation shows gFPca provides better schedulability than existing schedulers in general

• Future work: extensive evaluations using real CPS apps.
Results

- Generate task sets that favors nFPca
- Generate task sets that oppose to nFPca

(a) Generic workload
   # CPs of a task is uniformly distributed

(b) nFPca-favor workload
   Task with smaller period has smaller #CPs

(c) nFPca-oppose workload
   Task with smaller period has larger #CPs

**gFPca** is NOT always better than **nFPca**

**gFPca** has more improvement for nFPca-oppose workloads
Our overhead-aware analysis is substantially tighter
Thank You!
Questions?