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

Meng Xu  
*University of Pennsylvania*, mengxu@cis.upenn.edu

Linh Thi Xuan Phan  
*University of Pennsylvania*, linhphan@cis.upenn.edu

Hyon-Young Choi  
*University of Pennsylvania*

Insup Lee  
*University of Pennsylvania*, lee@cis.upenn.edu

Follow this and additional works at: [http://repository.upenn.edu/cis_papers](http://repository.upenn.edu/cis_papers)  
Part of the [Computer Engineering Commons](http://repository.upenn.edu/cis_papers) and the [Computer Sciences Commons](http://repository.upenn.edu/cis_papers)

**Recommended Citation**  

*IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*, Vienna, Austria, April 11-14, 2016  

This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_papers/816](http://repository.upenn.edu/cis_papers/816)  
For more information, please contact repository@pobox.upenn.edu.
Analysis and Implementation of Global Preemptive Fixed-Priority Scheduling with Dynamic Cache Allocation

Abstract
We introduce gFPca, a cache-aware global pre-emptive fixed-priority (FP) scheduling algorithm with dynamic cache allocation for multicore systems, and we present its analysis and implementation. We introduce a new overhead-aware analysis that integrates several novel ideas to safely and tightly account for the cache overhead. Our evaluation shows that the proposed overhead-accounting approach is highly accurate, and that gFPca improves the schedulability of cache-intensive tasksets substantially compared to the cache-agnostic global FP algorithm. Our evaluation also shows that gFPca outperforms the existing cache-aware non-preemptive global FP algorithm in most cases. Through our implementation and empirical evaluation, we demonstrate the feasibility of cache-aware global scheduling with dynamic cache allocation and highlight scenarios in which gFPca is especially useful in practice.

Keywords
cache storage, multiprocessing systems, processor scheduling, resource allocation, Dynamic scheduling, Heuristic algorithms, Interference, Multicore processing, Resource management, Scheduling algorithms, cache-aware global preemptive fixed-priority scheduling algorithm, cache-agnostic global FP algorithm, cache-aware nonpreemptive global FP algorithm, dynamic cache allocation, gFPca, multicore systems, overhead-aware analysis

Disciplines
Computer Engineering | Computer Sciences

Comments
IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016), Vienna, Austria, April 11-14, 2016

IEEEExplore page
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7461322

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/816
Abstract—We introduce gFPca, a cache-aware global preemptive fixed-priority (FP) scheduling algorithm with dynamic cache allocation for multicore systems, and we present its analysis and implementation. We show that a naive extension of existing overhead analysis techniques can lead to unsafe results, and we introduce a new overhead-aware analysis that integrates several novel ideas to safely and tightly account for the cache overhead. Our evaluation shows that the proposed overhead-accounting approach is highly accurate, and that gFPca not only improves schedulability of cache-intensive tasksets substantially compared to the cache-agnostic global FP algorithm but also outperforms the existing cache-aware non-preemptive global FP algorithm in most cases. Through our implementation and experimental evaluation, we demonstrate the feasibility of cache-aware global scheduling with dynamic cache allocation and highlight scenarios in which gFPca is especially useful in practice.

I. INTRODUCTION
Multicore processors are becoming pervasive, and it is becoming increasingly common to run real-time systems on a multicore platform. Most modern multicore platforms support a shared cache between the cores and the memory to deliver better hit rates and faster memory access latency. Although the shared cache can help increase the average performance, it also makes the worst-case timing analysis much more challenging due to the complex inter-core shared-cache interference: when tasks running simultaneously on different cores access the memories that are mapped to the same cache set, they may evict each other’s cache content from the cache, resulting in cache misses that are hard to predict.

One effective approach to bounding the inter-core cache interference is cache partitioning, which can be done using mechanisms such as page coloring [17] or way partitioning [23]. The idea is to divide the shared cache into multiple cache partitions and assign them to different tasks, such that tasks running simultaneously on different cores always use different cache partitions. Since tasks running concurrently never access one another’s cache partitions in this approach, the cache interference due to concurrent cache accesses can be eliminated, thus reducing the overall cache overhead and improving the worst-case response times of the tasks.

Cache partitioning has recently been explored in the real-time scheduling context. Most existing work in this line uses a static allocation for both cache and CPU resources [8, 18, 25], where cache partitions and tasks are assigned to specific cores offline. While this approach transforms the multicore analysis into the single core analysis, it can significantly underutilize resources because both the CPU and cache resources of one core may be left idle while another core is overloaded.

An alternative is to use cache-aware global scheduling, which dynamically allocates CPU and cache resources to tasks. At run time, each executing task locks all cache partitions it requires, so that the tasks running simultaneously on other cores cannot interfere with its cache content, and tasks can migrate among cores to better utilize the system resources. Guan et al. [15] has proposed a cache-aware non-preemptive fixed-priority scheduling algorithm with dynamic task-level cache allocation, which we will refer to as nFPca. Since nFPca does not allow preemptions, the schedulability analysis can be simplified; however, the non-preemptive nature can also lead to increased response times for high-priority tasks and undesirable priority inversions. In addition, the work in [15] does not provide any implementation of this algorithm.

In this paper, we investigate the feasibility of global preemptive scheduling with dynamic job-level cache allocation. We present gFPca, a cache-aware variant of the global preemptive fixed-priority (gFP) algorithm, together with its analysis and implementation. gFPca allocates cache to jobs dynamically at run time when they begin or resume, and it allows high-priority tasks to preempt low-priority tasks via both CPU and cache resources. It also allows low-priority tasks to execute when high-priority tasks are unable to execute due to insufficient cache resource, thus further improving the cache and CPU utilizations. Since preemption is allowed, tasks may experience cache overhead—e.g., upon resuming from a preemption, a task may need to reload its cache content in the cache partitions that were used by its higher-priority tasks; therefore, we develop a new method to account for such cache overhead.

The overhead accounting for gFPca is highly challenging, due to the extra resumption and preemption events that are not normally present in existing algorithms. To illustrate this, let us consider a dual-core system with three tasks: highest-priority $\tau_1$, medium-priority $\tau_2$, and low-priority $\tau_3$. Since each task may need different numbers of cache partitions to execute, under gFPca it is possible that $\tau_1$ and $\tau_3$ can run concurrently, whereas $\tau_2$ can only run alone. In this scenario, $\tau_3$ can be preempted not only when $\tau_2$ is released (while $\tau_1$ is having no job to execute) but also when $\tau_3$ completes its execution (which enables $\tau_2$ to resume its execution and thus preempt $\tau_1$). Similarly, the suspended task $\tau_3$ can resume not only when $\tau_2$ finishes but also when $\tau_1$ is released (which preempts $\tau_2$ and frees enough cache space for $\tau_1$ to continue). Due to this behavior, naive extensions of existing overhead accounting techniques can lead to unsafe analysis results for gFPca.

To tackle this challenge, we propose a new approach to safely and tightly account for the cache overhead, and then derive an overhead-aware schedulability analysis, for gFPca. The novelty of our approach lies in an integration of various strategies for overhead accounting: considering the combined effects of the source events that cause overhead, to mitigate potential double-accounting; exploiting the necessary condi-

*The published version of this paper contains a typo in Table 2 that has been fixed in this author version.
tions of task-preemption events with respect to cache and core configurations, to avoid accounting for the overhead that does not actually happen; and incorporating the scheduling behavior when bounding the overhead. Our evaluation shows that the new overhead accounting approach is very close to the best possible accounting approach.

In summary, we make the following specific contributions:
1. the gFPca scheduling algorithm (Section IV);
2. an implementation of gFPca, as well as nFPca and gFP, on an existing multicore hardware platform. (Section V).
3. an overhead-free analysis for gFPca (Section VI);
4. a demonstration that a naive extension of existing methods can lead to invalid results for gFP (Appendix B);
5. an overhead accounting approach and an overhead-aware schedulability analysis for gFPca (Section VII).

Our evaluation shows that our proposed overhead accounting approach is highly accurate. Further, gFPca improves schedulability substantially compared to the cache-agnostic gFP for cache-intensive workloads, and it outperforms nFPca for most cases in our experiments. Through our implementation and empirical evaluation that compared various scheduling strategies, we demonstrate the feasibility of cache-aware global scheduling with dynamic (job-level) cache allocation on real hardware platforms, and we highlight scenarios in which gFPca is especially needed in practice.

To the best of our knowledge, this work is the first to address the shared-cache overhead accounting for global preemptive fixed-priority multicore scheduling with dynamic cache allocation, and to provide an implementation of such an algorithm.

II. RELATED WORK

Several approaches for bounding the cache overhead on uniprocessor platforms have been developed (e.g., [2]), which integrate static cache analysis into the schedulability analysis. Our cache-aware analysis leverages these existing approaches; however, we show that a naive extension of these approaches may lead to invalid results for gFPca, and we present several ways to tackle the additional challenges in the global scheduling setting with dynamic cache allocation.

The shared cache overhead on multicore platforms has been considered in the context of WCET analysis, such as [14, 16, 21]. However, this line of work focuses on intrinsic cache overhead, and it does not consider the extrinsic cache overhead that arises due to scheduling, which our work addresses.

Scheduling algorithms that aim to reduce the cache effects on multicore platforms have also been investigated. For instance, Anderson et al. [4, 5, 13] proposed several heuristics of co-scheduling the tasks that share the same cache content to improve the cache performance while meeting real-time constraints. However, the WCETs of tasks with shared-cache overhead are assumed to be given a priori in these approaches. Although experiments show that the improved cache performance can reduce the average execution cost, the question of how to bound the shared-cache overhead and to derive a safe but tight WCET for each task is not addressed in their approaches (or existing work). In contrast, the cache overhead due to shared cache interference is not given a priori in the task WCET but computed in our analysis.

Schedulability analysis methods that consider the cache preemption or migration overhead have been proposed [6, 12, 19, 27, 30, 32]. These methods focus on the cache overhead associated with the loss of cache affinity as a result of a preemption or a migration. However, they do not consider the inter-core shared-cache interference between co-executing tasks: tasks running concurrently on different cores can still pollute the cache content of each other without invoking any preemption or migration event. Our work eliminates this type of shared cache interference via a combination of cache partitioning and cache-aware scheduling, and our analysis accounts for the overhead under this approach.

There exist shared cache management techniques for multicore platforms; for instance, Ward et al. [29] proposed cache locking and cache scheduling mechanisms to manage the shared cache for partitioned rate-monotonic scheduling. These mechanisms also support dynamic cache allocation to tasks, but they assume that tasks are partitioned onto cores. In contrast, gFPca schedules tasks globally on the cores, and thus it provides more scheduling flexibility while also presents new challenges to the overhead analysis.

A number of shared-cache partitioning mechanisms have also been proposed to reduce the shared-cache interference [8, 18, 24, 25]. Existing work typically considers a partitioned scheduling, which statically allocates tasks to different partitions of the shared cache and then to different cores. While this approach reduces the multicore scheduling problem to the single core scheduling problem, it cannot always be feasibly applied in practice (as we show in our empirical evaluation); for instance, when a taskset does not fit in the whole cache at the same time, different tasks are allowed to share the same cache partition over time and thus may still pollute each other’s cache content, which can result in deadline misses. Our work bridges this gap using a global scheduling approach with a dynamic allocation of cache partitions to tasks, while also accounting for the overhead in the analysis.

The only existing work we are aware of that considered global scheduling with dynamic cache allocation is Guan et al. [15], which proposed a cache-aware global non-preemptive fixed-priority scheduling algorithm (nFPca). Since preemptions are not allowed, tasks are always executed until completion and thus do not incur extrinsic cache overhead; therefore, the analysis in [15] is effectively an overhead-free analysis. However, this non-preemptive nature can lead to undesirable priority inversions and high response times for high-priority tasks. Our work provides an alternative using preemptive scheduling and dynamic job-level cache allocation, and the key new contributions compared to [15] lies in the novel approach to account for the overhead in this setting, as well as an implementation (of both gFPca and nFPca) on real hardware.

III. SYSTEM MODEL

We consider a multi-core platform with \( M \) identical cores and a shared cache that is accessible by all cores. The cache is partitioned into \( A \) equal cache partitions; we achieved this using the way partition mechanism [23]. The latency of

---

1. We only make this claim with respect to the overhead accounting approach but not the schedulability analysis.
reloading one partition is upper bounded by the maximum cache partition reload time, denoted by PRT. The value of PRT can be derived from the number of cache lines per partition and the maximum reloading time of one cache line.

The system consists of a set of independent explicit-deadline sporadic tasks, \( \tau = \{ \tau_1, \ldots, \tau_n \} \). Each task \( \tau_i \) is defined by \( \tau_i = (p_i, e_i, d_i, A_i) \), where \( p_i \), \( e_i \) and \( d_i \) are the minimum inter-arrival time (which we refer to as the period), worst-case execution time (WCET) and relative deadline of \( \tau_i \), and \( A_i \) is the number of cache partitions that \( \tau_i \) can use. Note that although the number of partitions allocated to \( \tau_i \) is fixed, under our scheduling approach, the exact partitions allocated to each job of \( \tau_i \) may change whenever it begins its execution or resumes from a preemption. We require that \( 0 < e_i \leq d_i \leq p_i \) and \( A_i \leq A \) for all \( \tau_i \in \tau \). Each task has a fixed priority \( \tau \), without loss of generality, we assume the tasks in \( \tau \) are sorted by their priorities, i.e., \( \tau_i \) has higher priority than \( \tau_j \) iff \( i < j \).

**Cache-related overhead.** When two code sections are mapped to the same cache line, one section can evict the other section’s cache line from the cache, which causes a cache miss when the task reschedules. If the two code sections belong to the same task, then the cache miss is an **intrinsic** cache miss; otherwise, it is an **extrinsic** cache miss [7]. The overhead (of reloading the evicted cache content) due to intrinsic cache misses of a task can typically be statically analyzed based on the task; however, extrinsic cache misses depend on the interference between tasks during their executions.

We assume that the WCET of each task already includes intrinsic cache-related overhead, and we focus on the extrinsic cache overhead. By abuse of terminology, throughout the paper, we refer to one **cache overhead** of a task as the time the task takes to reload its evicted cache content when it resumes from a preemption, and **total cache overhead** of a task as the total amount of time the task takes to reload its evicted cache content throughout the execution of a job of the task. We assume that the operating system does not affect the shared cache state of tasks, and we consider only the shared cache overhead in this paper and defer the incorporation of the private cache overhead to future work.

**ECP and UCP.** As defined in [28], an Evicting Cache Block (ECB) of a task is a cache line that the task can access during its execution. Likewise, we define an **Evicting Cache Partition** (ECP) of a task to be a cache partition that the task can access. We denote by ECP\(_k\) the set of ECPs of \( \tau_k \) during an uninterrupted execution interval of \( \tau_k \); note that ECP\(_k\) varies across different continuous execution intervals of \( \tau_k \), but \( |\text{ECP}_k| \leq A_k \) by definition. In addition, we define a **Useful Cache Partition** (UCP) of \( \tau_k \) to be a cache partition that \( \tau_k \) accesses at some time point and later accesses again as cache hit, when \( \tau_k \) executes alone in the system (similar to UCB notion [3]). The set of UCPs of \( \tau_k \) is denoted by UCP\(_k\).

---

### IV. gFPca Scheduling Algorithm

We now present the gFPca algorithm. Like gFP, gFPca also schedules tasks based on their (fixed) priorities; however, a task is only executed if there are sufficient cache partitions for it (including also the partitions obtained by preempting one or more lower-priority tasks), and low-priority tasks can execute if all pending high-priority tasks are unable to execute.

Specifically, gFPca makes scheduling decisions whenever a task releases a new job or finishes its current job’s execution (or is blocked or unblocked via resources other than cache and CPU). At each scheduling point, it tries to schedule pending tasks in decreasing order of priority. For each pending task \( \tau_i \):

- **Step 1)** First, gFPca looks for an idle core; if none exists, it considers the core that is executing the lowest-priority task among all currently executing tasks with lower priority than \( \tau_i \), if such tasks exist. If no such core is found, it returns.

- **Step 2)** Next, gFPca tries to find \( A_i \) cache partitions for \( \tau_i \), considering the idle partitions first and then the partitions obtained by preempting \( \tau_i \)'s lower-priority tasks (chosen in increasing order of priority). If successful, it will reserve those \( A_i \) partitions \( \tau_i \) to preempt the lower-priority tasks that are using those partitions or using the core chosen in Step 1, and schedule \( \tau_i \) to run on the chosen core. Otherwise, gFPca will move to the next task and repeat the process from Step 1.6

Observe that under gFPca, cache partitions are allocated to each job **dynamically** at run time when it begins its execution and when it resumes. Whenever this occurs, the system maps some or all of the memory accesses of the task to the allocated partitions (which may include those previously belonged to a preempted task). When a preempted task resumes, it needs to reload its information from the memory to the cache, if this information has been polluted by higher-priority tasks or if it is assigned new cache partitions. Our implementation does not require any memory page copy for reassigning partitions, and our analysis considers the costs of mapping the memory accesses and reloading the memory content into the cache.

### V. IMPLEMENTATION

We implemented gFPca within LITMUS\textsuperscript{RT} on the Freescale LMX6 quad-core evaluation board, which supports way partitioning through the PL310 cache controller. For comparison, we also implemented the existing non-preemptive nFPca in [15] and the cache-agnostic gFP schedulers.

#### A. Dynamic cache control

We utilized the **Lockdown by Master** (LbM) mechanism, supported by the PL310 controller, for our cache allocation (using a similar approach as [23, 29]). The LbM allows certain ways to be marked as unavailable for allocation, such that the cache allocation (which allocates cache lines for cache misses) only happens in the remaining ways that are not marked as unavailable. Each core \( P_i \) has a per-CPU lockdown register \( R_i \)\(^7\), where a bit \( q \) in \( R_i \) is one if the cache allocation cannot

---

\(^2\)Different values of \( A_i \) may lead to different values of \( e_i \); our analysis holds for any given value of \( A_i \) (and corresponding \( e_i \)). For our numerical evaluation, \( A_i \) was chosen to be the smallest number of cache partitions that leads to the minimum WCET for \( \tau_i \) (when it executes alone).

\(^3\)Ties among tasks with the same priority are broken based on their indices, where a task with a smaller index has a higher priority.

\(^4\)One way to avoid the shared cache interference between the OS and tasks is to dedicate a specific area of the cache to the OS.

\(^5\)When more than \( A_i \) partitions are found, gFPca gives preference to the ones that still hold the cache content of the task \( \tau_i \).

\(^6\)gFPca imposes no constraints among the partitions allocated to a task; however, both its cache allocation and analysis can easily be modified to incorporate potential constraints, e.g., one that imposes contiguous partitions.

\(^7\)To be precise, each core has two separate registers for instruction and data access, but we focus on data access in this paper.
components in LITMUS\textsuperscript{RT} to incorporate gFPca’s cache management and scheduling behavior. Most notable extensions include: (1) \textit{RT Task:} We extended the \texttt{rt_params} field, which holds the timing information of a real-time task, with the cache information (i.e., the number of cache partitions, the set of currently used partitions, and the set of previously used partitions). (2) \textit{RT-Context:} We extended the \texttt{cpu_entry} data structure, which holds the real-time context of a core, with a new field called \texttt{preempting} to indicate whether the core is preempted via cache. (3) \textit{Scheduling real-time domain}, which holds all (global) information of the cores and real-time tasks, such as the release and ready queues (not shown in Fig. 1). We extended the scheduling domain to include two new components: CP-bitmap and CPtoTask-map. CP-bitmap is a bitmap that indicates whether a cache partition is locked (i.e., reserved for some task). CPtoTask-map maps each partition to a task that it belongs (if any). The architecture also includes the PL310 cache controller that controls the 16 cache partitions of the L2 shared cache. For synchronization, we used three global spin locks: one for the release queue; one for the ready queue, RT-Context, and CP-bitmap; and one for CPtoTask-map and the cache controller’s registers.

The \texttt{gFPca} scheduler: The steps in Fig. 1 illustrate how the scheduler on a core works in a nutshell. Specifically, when a scheduling event (task-release, task-finish, task-blocked on other resources such as I/O, or task-unblocked event) arrives at a core (e.g., P1), the scheduler on that core will be invoked. Once being invoked, the scheduler performs Steps 1–3:

Step 1) Executes the \texttt{check_for_preemption} function, which implements the gFPca algorithm (described in Section IV), to determine: the highest-priority ready task that can execute next, the core to execute the task, the cache partitions to reserve for the task, and the currently running tasks to be preempted. The scheduler then continues to the next highest-priority ready task, until no more ready task can be scheduled. For the example in Fig. 1, the scheduler on P1 decides to preempt the tasks currently running on P0 and P2 (say \(\tau_i\) and \(\tau_j\), respectively) and schedule the ready task (say \(\tau_k\)) on P0.

Step 2) Updates CP-bitmap to reflect the new locked cache partitions, and updates the RT-Context of the preempted cores and the core(s) that will run the scheduled tasks. In Fig. 1, P1’s scheduler modifies CP-bitmap by unmarking the cache partitions that were assigned to \(\tau_i\) and \(\tau_j\) and then marking the partitions that will be reserved for \(\tau_k\). In addition, it updates P0’s \textit{linked task} (i.e., the real-time task to execute next) to be \(\tau_i\), P2’s \textit{linked task} to be NULL and P2’s \textit{preempting} field to be true (to indicate that P2 is preempted via cache only).

Step 3) Sends an Inter-Processor Interrupt (IPI) to each preempted core and each core that will run a scheduled task, to notify the preempted core to preempt its currently running task and the scheduled core to execute its linked task (e.g., P0 to preempt \(\tau_i\) and run \(\tau_j\), and P2 to preempt \(\tau_j\)).

When a core receives the above IPI, the scheduler on that component...

---

\textbf{B. Scheduling architecture}

Fig. 1 shows a high-level overview of the scheduling architecture for gFPca. Our implementation extended various components in the LITMUS\textsuperscript{RT} architecture for...
core will be invoked, and it will perform the next three steps:
Step 4) Moves the linked task (configured in Step 2) to the
core, and updates the scheduled task of the core to be the
linked task. (If the linked task is NULL, the scheduler will
pick a non-real-time task to execute on the core.)
Step 5) Determines which of the cache partitions reserved
for the linked task should be flushed (i.e., if used by other
tasks), flushes those partitions, and updates CPtoTask-map to
reflect the new mapping of partitions to tasks.
Step 6) Starts executing the linked task.

C. Run-time overhead
We used the feather-trace tool (with a small modification)
to measure the run-time overhead under gFPca and nFPca
schedulers, and the existing gEDF scheduler in LITMUS\textsuperscript{R7}.
We observed that both gFPca and nFPca schedulers incur
similar average release, scheduling, and IPI delay overheads as
gEDF does. However, they have larger average context switch
overhead; this is due to potential cache flush during a context
switch. The gFPca scheduler incurs higher worst-case over-
heads than the gEDF scheduler, which is expected because the
gFPca algorithm has a higher complexity. (Additional details
are described in Appendix A.)

In the coming sections, we present the schedulability anal-
ysis of gFPca, first assuming the absence of overhead and
then considering the overhead. The overhead-aware analysis
focuses on cache-related preemption and migration delay
(CRPMD) overhead (as this is most challenging), but can
easily be extended to include other overheads (see Appendix
D), and our evaluation considered all these overheads. Due
to space constraints, we omit the proofs here, but they are
available in [31].

VI. OVERHEAD-FREE ANALYSIS
The overhead-free schedulability analysis of gFPca can be
established using a similar idea as that of nFPca [15]. As
usual, the demand of a task \( \tau_i \) in any time interval \([a, b]\) is
the maximum amount of computation that must be completed
within \([a, b]\) to ensure that all jobs of \( \tau_i \) with deadlines within
\([a, b]\) are schedulable. When \( \tau_i = (p_i, e_i, d_i, A_i) \) is scheduled
under gFPca, \( \tau_i \) has the maximum amount of computation
in a period of another task \( \tau_k \) when the first job of \( \tau_k \) starts
executing at the release time of \( \tau_i \) and the following jobs of
\( \tau_k \) execute as early as possible, as illustrated in Fig. 2. Hence,
the worst-case demand of \( \tau_i \) in a period of \( \tau_k \) is given by [9]:

\[
W_i^k = NJ_i^k \times e_i + \min\{d_k + d_i - e_i - NJ_i^k \times p_i, e_i\},
\]

(1)

where \( NJ_i^k = \left\lfloor \frac{d_k + d_i - e_i}{p_i} \right\rfloor \) is the maximum number of jobs of \( \tau_i \)
that have the entire executions falling within a period of \( \tau_k \).

Fig. 2: Worst-case demand of \( \tau_i \) in a period of \( \tau_k \) scenario.

The length of \( \tau_k \)'s busy interval, denoted by \( B_k \), is the total
length of all subintervals in a period of \( \tau_k \) during which it
cannot execute. The busy interval of \( \tau_k \) can be grouped into
two categories: (1) CPU-busy interval, during which all cores
are busy executing other higher-priority tasks; and (2) cache-
busy interval, during which at least one core is available\textsuperscript{10} and
at least \( A - A_k + 1 \) cache partitions are assigned to \( \tau_k \)'s higher-

10A core is available to \( \tau_i \) if it is idle or executing a lower-priority task.

11To obtain \( \hat{B}_k \), the solver of the LP problem has to calculate the values of
\( \alpha_i^k \) and \( \beta_i^k \) for each higher-priority task \( \tau_i \).
\( \tau_j \) \( (i < j < k) \) is blocked due to insufficient cache for it; when \( \tau_i \) finishes, \( \tau_k \) is preempted by \( \tau_j \) because there is now sufficient cache for \( \tau_j \) to execute. Due to this behavior, existing approaches cannot be applied, and naïve extensions may lead to unsafe results for gFPca (see Appendix B for an example).

Our idea is to account for the overhead by analyzing the source events that cause cache overhead, and analyze the combined total overhead they cause to a task. As not every task experiences (extrinsic) overhead, e.g., the highest-priority task, we also derive the necessary conditions under which a task may experience overhead. Specifically, we first identify the cache-related task events and establish the necessary conditions under which these events cause a task to experience overhead. These conditions are then used to derive the set of tasks that may preempt a task \( \tau_k \) via CPU or cache resource. Finally, we analyze the total overhead of \( \tau_t \) that is caused by the cache-related events of other tasks and include it into \( \tau_t \)'s WCET, then we apply the overhead-free schedulability analysis on the inflated taskset. (For simplicity, we will simply write ‘overhead’ in place of ‘cache overhead’.)

A. Cache-related task events

Under gFPca, the system has five types of task events: task-release, task-finish, task-preemption, task-resumption, and task-migration events. Because the cache is shared by all cores, no overhead is incurred when a task migrates from one core to another; therefore, a task-migration event of a task does not lead to any overhead and we only need to consider the other four types of task events.

A task-preemption event of \( \tau_k \) occurs when the CPU or cache resource allocated to \( \tau_k \) is reduced. Because new jobs are released when task-release events occur and existing jobs resume when task-resumption events occur, a higher-priority task \( \tau_j \) with the task-release or task-resumption event may take the CPU and/or cache resource from \( \tau_k \), thus leading to a task-preemption event of \( \tau_k \). Similarly, because running jobs may stop at task-preemption and task-finish events, and the released CPU or cache resource may be allocated to \( \tau_k \), both task-preemption and task-finish events of \( \tau_j \) may lead to a task-resumption event of \( \tau_k \). Further, a task-preemption event may lead to a task-resumption event and vice versa.

If the arrival of a task event \( A \) may lead to the arrival of another task event \( B \), then we say \( A \) causes \( B \), denoted as \( A \rightarrow B \). The causal relations of task events are illustrated in Fig. 3. It is clear from the figure that the task-release and task-finish events are the root causes of the other events. Since a task experiences overhead only at its task-resumption events, which are caused by task-release and task-finish events of other tasks, if the task-release and task-finish events are eliminated, the overhead will also be eliminated.

Lemma 3. Task-release events and task-finish events are the source events that cause overhead in a system.

Based on Lemma 3, if we can compute a bound on the overhead that each task-release event and each task-finish event of a higher-priority task \( \tau_i \) cause to a lower-priority task \( \tau_k \), then we can safely account for the total overhead of \( \tau_k \). To derive this bound, we will analyze the set of tasks that can preempt \( \tau_k \) based on the necessary conditions of task-preemption events, which we now establish.

B. Conditions of task-preemption events

The overhead a task \( \tau_t \) experiences come from its preemption-events, which are caused by the task-release and task-finish events of its higher-priority tasks. A higher-priority task \( \tau_i \) may preempt \( \tau_k \) via either CPU and/or cache resources; however, no task-preemption event of \( \tau_t \) occurs if the number of cores is larger than the number of tasks in the system and the number of cache partitions of the platform is sufficient for all tasks. The next lemmas state the conditions of a preemption via CPU and cache resources, respectively.

**Lemma 4.** If a task \( \tau_i \) preempts a task \( \tau_k \)'s CPU resource at time \( t \), then \( \tau_i \) must have higher priority than \( \tau_k \) and the number of tasks with higher priority than \( \tau_i \) must be at least the total number of cores in the system, i.e., \( \sum_{j<k} 1 \geq M \).

**Lemma 5.** If \( \tau_i \) preempts \( \tau_k \)'s cache resource at time \( t \), then \( \tau_i \) must have higher priority than \( \tau_k \) and the total number of cache partitions of \( \tau_i \) with \( j < k \) must be larger than \( A - A_k \), where \( A \) is the number of cache partitions of the cache.

Let \( \rho_k \) and \( \kappa_k \) be the maximum sets of tasks that may preempt \( \tau_k \) via CPU and cache resources, respectively. Due to the above lemmas, we have:

\[
\rho_k = \{ \tau_i | i < k \text{ and } \sum_{j<k} 1 \geq M \} \\
\kappa_k = \{ \tau_i | i < k \text{ and } \sum_{j<k} A_j > A \}
\]

As a result, the set of tasks that may preempt \( \tau_k \) via either CPU or cache or both resources is \( \rho_k \cup \kappa_k \).

C. Overhead caused by a task-release event

Based on the established conditions of a task-preemption event of \( \tau_k \), we can analyze the overhead of \( \tau_k \) that is caused by one task-release event of a higher-priority task \( \tau_i \).

Observe that when \( \tau_k \) releases a job at time \( t_1 \), the cache partitions \( \tau_i \) may access and pollute are in \( \text{ECP}_i \). If \( \tau_k \) is preempted at the task-release event of \( \tau_i \), \( \tau_k \) can directly evict all cache partitions in \( \text{ECP}_i \) that \( \tau_k \) may use in the worst case.

Further, another higher-priority task \( \tau_j \) of \( \tau_k \) may release a job at time \( t_1 \) as well. Although such a task-release event may also cause overhead to \( \tau_k \), this overhead will be considered as the overhead caused by \( \tau_j \)'s task-release events (rather than by \( \tau_k \)'s). Further, under gFPca, a lower-priority task \( \tau_i \) may also pollute the cache partitions of \( \tau_k \) while \( \tau_k \) is being preempted due to a task-release event of \( \tau_i \). However, not every lower-priority task \( \tau_j \) can pollute the cache partitions of \( \tau_k \).

**Lemma 6.** When a release-event of \( \tau_i \) occurs, if \( \tau_k \) is preempted but a lower-priority task \( \tau_j \) \( (k < l) \) either resumes from a preemption or releases a new job and this job is executed, then the number of cache partitions of \( \tau_i \) must be less than that of \( \tau_k \), i.e., \( A_j < A_k \).

Let \( \phi_k^l \) denote the set of useful cache partitions of \( \tau_k \) that may be polluted due to a task-release event of \( \tau_i \). When a
task-release event of $\tau_i$ occurs, there are three scenarios: (1) $\tau_i$ does not preempt $\tau_k$ (as there are sufficient CPU and cache resources for $\tau_k$), in which case $\tau_k$ experiences no overhead due to this task-release event of $\tau_i$; (2) $\tau_i$ preempts $\tau_k$ by taking only $\tau_k$’s CPU resource, in which case only the lower-priority tasks of $\tau_i$ may pollute the UCPs of $\tau_k$; and (3) $\tau_i$ preempts $\tau_k$ by taking $\tau_k$’s cache resource, in which case both $\tau_i$ and lower-priority tasks of $\tau_i$ may pollute the UCPs of $\tau_k$. Therefore, $\phi_{i,k}^1$ can be calculated as follows:

$$\phi_{i,k}^1 = \begin{cases} 
\text{UCP}_{i} \cap (\text{ECP}_{j}\cup(k<\ell, \ell<A_i \land \ell\neq k)) & \text{if } t_i \leq \kappa_k \\
\text{UCP}_{i} \cap (k<\ell, \ell<A_i \land \ell\neq k) & \text{if } t_i \leq \kappa_k \wedge t_i \leq \rho_k \\
0 & \text{if } t_i \notin \{\kappa_k \cup \rho_k\}
\end{cases}$$

(4)

Given any two sets $S_1$ and $S_2$, we have $|S_1 \cup S_2| \leq |S_1| + |S_2|$ and $|S_1 \cap S_2| \leq \min\{ |S_1|, |S_2| \}$. Hence, we obtain:

$$|\phi_{i,k}^1| \leq \min\{ |\text{UCP}_{i} \cap \text{ECP}_{j}|, \sum_{k<l, \ell<A_i} |\text{ECP}_{j}| \}, \text{ if } t_i \notin \kappa_k \wedge t_i \leq \rho_k$$

(5)

Denote by $\Delta^i_{j,k}$ the overhead of $\tau_i$ that is caused by a task-release event of $\tau_k$, where $i < k$. Then,

$$\Delta^i_{j,k} \leq \text{PRT} \times |\phi_{i,j}^1|.$$  

(6)

**D. Overhead caused by a task-finish event**

When a task $\tau_i$ finishes its execution at time $t_2$, the overhead that task $\tau_k$ may experience due to this task-finish event falls into the following cases:

Case 1) $\tau_k$ is not running at $t_2$: If $\tau_k$ finishes before or at $t_2$, then clearly the task-finish event causes no overhead to $\tau_k$. If it has not finished its execution at $t_2$, this task-finish event also does not bring any overhead to $\tau_k$, because even though $\tau_k$ might have polluted $\tau_i$’s cache before $t_2$, the pollution is caused by other task-release or task-finish events of $\tau_i$ and should be accounted in the cost of those events.

Case 2) $\tau_k$ is running at $t_2$: If $\tau_k$ continues to run after $t_2$, then it incurs no overhead as it is not preempted. However, if $\tau_k$ is preempted at $t_2$, then it must be preempted by another higher-priority task $\tau_j$ that is resumed at $t_2$ when $\tau_k$ finishes, in which case $\tau_j$ can access and pollute any cache partitions in ECP. However, as stated in the next two lemmas, at most one task $\tau_j$ with $i < j < k$ can resume and preempt $\tau_k$ at $t_2$, and the number of cache partitions this task can access should be more than that of $\tau_j$.

**Lemma 7.** If a task $\tau_j$, where $i < j < k$, resumes and preempts $\tau_k$ at a task-finish event of $\tau_i$, then $A_j > A_k$.

**Lemma 8.** There exists at most one task $\tau_j$ with $i < j < k$ that can resume and preempt $\tau_k$ at a task-finish event of $\tau_i$.

In addition, when $\tau_j$ is preempted, lower-priority tasks of $\tau_j$ may also resume or release new jobs and these jobs are executed, and thus they may pollute the cache partitions of $\tau_j$. According to Lemma 6, only lower-priority tasks $\tau_l$ with $k < l$ and $A_l < A_k$ may pollute $\tau_j$’s cache partitions while $\tau_j$ is being preempted. When a task $\tau_j$ ($i < j < k$) resumes and preempts $\tau_k$ at the occurrence of the task-finish event of $\tau_i$, the set of useful cache partitions of $\tau_k$ that may be polluted, denoted by $\phi_{i,j,k}$, is the same with the set of useful cache partition of $\tau_i$ that may be polluted at the task-release event of $\tau_i$. Therefore, $\phi_{i,j,k} = \phi_{i,j,k}$ and the size of $\phi_{i,j,k} = |\phi_{i,j,k}| = |\phi_{i,j,k}|$.

Let $\Delta^i_{j,k}$ denote the overhead of $\tau_i$ that is caused by a task-finish event of $\tau_j$, where $i < k$. Since any task $\tau_j$ ($i < j < k$ and $A_j < A_i$) may resume and preempt $\tau_i$ at the task-finish event of $\tau_j$, we obtain

$$\Delta^i_{j,k} \leq \max_{i<j<k, A_k<A_i} \text{PRT} \times |\phi_{i,j,k}|.$$  

(7)

**E. Overhead-aware schedulability analysis**

In the previous sections, we have computed the maximum overhead that each task-release event and each task-finish event of a higher-priority task $\tau_i$ causes to a lower-priority task $\tau_k$. To account for the overall overhead $\tau_k$ experiences, we need to compute the number of task-release and task-finish events of higher-priority tasks in each period of $\tau_k$.

Since each job of a task has one task-release event and one task-finish event, it may seem at first that an upper bound on the total number of task-release and task-finish events of all higher-priority tasks in the period of $\tau_k$ is $\sum_{i<k} 2^{|\phi_{i,k}|} + 2$. While this bound is safe, it is not tight because not every task-release or task-finish event of each job of higher-priority tasks can cause overhead to $\tau_k$, as stated by Lemma 9.

**Lemma 9.** Suppose the task-release and task-finish events of the same job of $\tau_i$ occur at time $t_1$ and $t_2$, respectively, and suppose the lower-priority task $\tau_k$ is preempted at $t_1$ and $t_2$ as well. Then, at least one task-release or task-finish event of another higher-priority task $\tau_j$ ($j < k$) must occur at time $t_3$, where $t_1 < t_3 < t_2$ and $\tau_j$ resumes at $t_3$.

Thus, instead of accounting for the overhead caused by each task-release and each task-finish event of higher-priority tasks, we account for the overhead of $\tau_k$ that is caused by each job of its higher-priority tasks in a period of $\tau_k$, as follows:

If only one of the task-release and task-finish events of the same job of $\tau_i$ may cause overhead to $\tau_k$, the overhead caused by each job of $\tau_i$ is $\Delta^i_{k,j} = \Delta^i_{k,j} - \min_{j<k,i \neq j}(\Delta^i_{j,k}, \Delta^i_{j,k})$. Hence, the overhead of $\tau_k$ that is caused by one job of a higher-priority task $\tau_i$ is bounded by

$$\delta^i = \max\{\Delta^i_{k,j}, \Delta^i_{k,j} + \Delta^i_{k,j} - \min_{j<k,i \neq j}(\Delta^i_{j,k}, \Delta^i_{j,k})\}.$$  

Further, the number of jobs of $\tau_i$ in a period of $\tau_k$ that have both release and finish events causing $\tau_k$ to resume is at most $N_i^k = \left\lceil \frac{\text{PRT}}{\text{PRT}} \right\rceil$. Since the finish event of the carry-in job of $\tau_j$ and the release event of the carry-out job of $\tau_j$ in a period of $\tau_k$ may also lead to one task-resumption event of $\tau_k$, we imply that the overhead of $\tau_k$ that is caused by all of its higher-priority tasks is upper bounded by

$$\delta^k = \sum_{i=1}^{k-1} \delta_i^k \times N_i^k + \Delta^i_{k,j} + \Delta^i_{k,j}.$$  

(8)
The overhead-aware analysis can now be done by first inflating the WCET of each task \( \tau_k \) with \( \delta_k \), and then applying the overhead-free analysis (Section VI) on the inflated taskset.

**Theorem 10.** A taskset \( \tau = \{ \tau_1, \ldots, \tau_n \} \), where \( \tau_k = (p_k, e_k, d_k, A_k) \), is schedulable under \( gFPca \) in the presence of cache overhead if \( \tau' = \{ \tau'_1, \ldots, \tau'_n \} \) satisfies Theorem 1, where \( \tau'_k = (p_k, e'_k, d_k, A_k) \) and \( e'_k = e_k + \delta_k \) for all \( 1 \leq k \leq n \).

**VIII. NUMERICAL EVALUATION**

Our evaluation was based on randomly generated real-time workloads and our implementation platform, which has four cores and a 1MB shared cache that is partitioned into 16 equal partitions. We had two main objectives: (1) Evaluate the accuracy of the overhead-aware analysis for \( gFPca \), by comparing to the overhead-free analysis and a baseline overhead-aware analysis; intuitively, the closer the overhead-aware schedulability results are to the overhead-free schedulability results, the closer the overhead accounting is to an optimal overhead accounting method. (2) Investigate the performance of \( gFPca \) in comparison to \( gFP \) and \( nFPca \).

For the baseline, since no existing overhead-aware analysis can be directly applied to \( gFPca \), we used an extension of existing approach (which is safe for \( gFPca \)) that works as follows: first inflates the WCET of each task \( \tau_i \) (\( i > 1 \)) with the total overhead it experiences during an entire execution of a job, and then applies \( gFPca \)'s overhead-free analysis.

**Types of overhead.** Besides CRPMD overhead, our evaluation considered four other types: release, scheduling, IPC, and context switch. For this, we extended the analysis in Section VII, in [15], and in [9], to account for all overhead types under \( gFPca \), \( nFPca \), and \( gFP \), respectively (extension details are in the appendix). We measured the overhead values of each scheduler in our implementation.

**Workload.** Each workload contained a set of randomly generated implicit-deadline sporadic task sets. The tasks’ utilizations followed one of four distributions: a uniform distribution within the range \( [0.5, 0.9] \) and three bimodal distributions, where the utilizations were uniformly distributed in either \( [0.001, 0.5] \) or \( [0.5, 0.9] \) with respective probabilities of 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy), as used in [30, 32]. The number of ECPs of a task was uniformly distributed in [1, 8] by default, and the number of UCPs was set equal to the number of ECPs.

**Overhead values.** For the CRPMD overhead, the latency of reloading one cache line measured on our board was 90.89ns. The size of each cache line is 32B, and thus each cache partition has \( \frac{1MB}{32B} \times 16 = 2048 \) cache lines. Hence, it takes at most 90.89ns \( \times 2048 \leq 0.19 \)ms to reload one cache partition. Hence, we set the cache partition reloading time \( PRT = 0.19 \)ms.

We measured the remaining overheads for each scheduler (\( gFPca \), \( nFPca \), \( gFP \)), and used monotonic piece-wise linear interpolation to derive the upper-bounds of each overhead under each scheduler as a function of the taskset size. For \( gFPca \), the context switch overhead also includes the overhead for (re)assigning cache partitions, which we derived from the measured maximum latency of flushing one cache partition. (Details of the overhead values are available in [31]).

**A. Evaluation of the overhead-aware analysis**

We generated 500 tasksets with taskset utilization ranging from 0.1 to 4, with a step of 0.2. For each taskset utilization, we generated 85 independently generated tasksets; the task utilizations were uniformly distributed in \([0.5, 0.9] \); the task periods were uniformly distributed in \([10, 400] ms \). We computed the fraction of schedulable tasksets under each analysis.

The results show that our overhead-aware analysis (shown as \( gFPca \)) is substantially tighter than the baseline; for example, when the taskset utilization is 2.7, the baseline analysis claimed that no taskset is schedulable, even though 60% of the tasksets are schedulable under our overhead-aware analysis. The results also show that the fractions of schedulable tasksets under our overhead-aware analysis and the overhead-free analysis are very close across all taskset utilizations. This means that our overhead-accounting technique is very close to an optimal overhead-accounting technique, which can be explained from its novel strategies for bounding the overhead.

We also evaluated the impacts of core and cache configurations, and the results further confirm these observations.

**B. Evaluation of \( gFPca \)'s performance.**

We generated 500 tasksets as before. The number of cache partitions of each task was uniformly distributed in \([1, 12] \). The period range that each task chooses was uniformly distributed in \([550, 650]\) (as in [20]). We analyzed the schedulability of each taskset under \( gFPca \), \( nFPca \), and \( gFP \).

**Cache access information for \( gFP \) analysis.** The overhead-aware analysis for \( gFP \) needs to consider the shared cache interference among concurrent tasks (which are eliminated in \( gFPca \) and \( nFPca \)). We derived the overhead that a task experiences from the cache hit latency (55.77ns), miss latency (146.66ns), and the \( hit_time_ratio \) of the task (i.e., the ratio of the time it spends on cache hit accesses to its execution time when executing alone). To generate different cache access scenarios, the \( hit_time_ratio \) of tasks was uniformly distributed in \([0.1, 0.3] \) (cache light), \([0.3, 0.6] \) (cache medium), and \([0.6, 0.9] \) (cache heavy). The generated \( hit_time_ratio \) values were then used for the analysis under \( gFP \).

Fig. 5 shows the fractions of schedulable tasksets under each algorithm. The lines with the labels \( gFP-H \), \( gFP-M \) and \( gFP-L \) represent the results under \( gFP \) for the cache light, cache medium, and cache heavy scenarios, respectively.

**Benefits of cache-aware scheduling:** As Fig. 5 shows, both \( gFPca \) and \( gFPca \) perform much better than the cache-agnostic \( gFP \) under the cache medium and cache heavy configurations, and for most taskset utilizations under the cache light configuration. This is expected, because \( gFP \) does not protect concurrently running tasks from cache interference, which is more obvious for more cache-intensive workloads. On the contrary, both \( gFPca \) and \( nFPca \) mitigate such interference via cache partitioning and cache-aware scheduling, and thus they can significantly improve the schedulability of the tasksets.

Comparing the fractions of schedulable tasksets under \( gFP \) when the \( hit_time_ratio \) of tasks is in the cache light, cache medium and cache heavy scenarios, we observe that as the \( hit_time_ratio \) of tasks increases, the performance of \( gFP \) decreases. One reason for this trend is that tasks with a
larger hit_time_ratio have more cache hit accesses when they execute alone, and hence they are more sensitive to the shared cache interference under gFP. Note that under gFP, we had to assume every cache hit access when it executes alone may be polluted by tasks running concurrently on other cores when it is scheduled with other tasks; therefore, a higher number of cache hit accesses leads to a larger extrinsic cache overhead.

**Benefits of gFPca over nFPca:** We observe that gFPca outperforms nFPca in terms of the fraction of schedulable tasksets across all but one taskset utilizations. This is because gFPca avoids undesirable priority inversions and allows low-priority tasks to execute if high-priority tasks are unable to, and thus it can better utilize the system’s resources.

**The number of cache partitions and task priority relation:**

Because nFPca does not allow lower-priority tasks to execute when any higher-priority task is blocked by cache resource, it performs better on tasksets in which higher-priority tasks require a smaller number of cache partitions and worse on tasksets in which higher-priority tasks require a higher number of cache partitions. To investigate the impact of the relation between the number of cache partitions and the task priority on the performance of the algorithms, we generated two kinds of tasksets: (1) the so-called nFPca-favor tasks, in which $|A_i| = \lfloor \frac{\tau_i}{\tau_{i-1} - \tau_{i-2}} \times 12 \rfloor$ for each task $\tau_i$, and (2) the so-called nFPca-oppose tasks, in which $|A_i| = \lfloor \frac{1}{\max_period - \min_period} \times 1 \rfloor$ for each $\tau_i$. Other parameters of the tasks were generated in the same manner as above.

The fractions of schedulable tasksets are shown in Fig. 6. On the nFPca-favor tasksets, nFPca performs better than gFPca but only slightly, although the tasksets favor nFPca. We attributed this to the work-conserving nature of gFPca, which allows it to better utilize the system’s resource. In contrast, the results in Fig. 7 show that gFPca can schedule many more tasksets than nFPca does on the nFPca-oppose tasksets. We can also observe that the performance improvement that gFPca achieves over nFPca increases as higher-priority tasks use more cache partitions (as the tasksets move from the nFPca-favor to the nFPca-oppose).

**IX. EMPIRICAL EVALUATION**

We used synthetic workloads to illustrate the applicability and benefits of gFPca based on our implementation platform (with four cores, 16 cache partitions). We focused on tasks that are sensitive to shared cache interferences (for which cache isolation is critical), and evaluated four algorithms: gFP (cache-agnostic global scheduling), pFP (partitioned scheduling), gFPca (cache-aware non-preemptive global scheduling with dynamic task-level cache allocation), and gFPca (cache-aware preemptive scheduling with dynamic job-level cache allocation).

**Workload generation.** We first constructed two real-time programs in our implementation: the first randomly accesses every 32 bytes (the size of a cache line) in a 960KB array for 200 times, which was used for the highest-priority task; and the second randomly accesses every 32 bytes in a 192KB array for 2000 times, which was used for each lower-priority task.

We separately measured the WCET of each program under the gFPca scheduler when it was allocated different numbers of cache partitions; the results are shown Fig. 8.

We then constructed a reference taskset $\tau_{max}$ with $n = 5$ tasks, with $\tau_1 > \tau_2 > \cdots > \tau_n$, where $\tau_i = (p_i = 5000, d_i = 500)$ and $\tau_i = (p_i = 5000, d_i = 1550)$ for all $1 < i < n$.14

**Fig. 4:** Analysis accuracy

**Fig. 5:** Generic.

**Fig. 6:** nFPca-favor.

**Fig. 7:** nFPca-oppose.

**Fig. 8:** Measured WCET vs. Number of cache partitions. Horizontal line indicates relative deadline.

**Analysis of WCET and the number of cache partitions.** Fig. 8 shows that the WCET of $\tau_1$ is 430ms with 16 cache partitions and 501ms with 15 cache partitions. Since its deadline is 500ms, $\tau_1$ needs all 16 cache partitions to meet its deadline. Each lower-priority task has a WCET of 800ms with 4 cache partitions, a WCET of 1059ms with 3 cache partitions and a WCET of 1958ms with 0 cache partition.

From the above analysis, we could feasibly assign the number of partitions of each task under gFPca and nFPca, i.e., $A_1 = 16$ and $A_4 = i$ ($i > 1$). We set the WCET of each task to be an upper bound of the WCET measured under the assigned number of partitions15, i.e., $e_i = 500$ and $e_i = 1050$; this was used in our experiment investigating the impact of task density. (Note that, these WCETs are safe under gFP as well, since gFP allows every task to access the entire cache.)

**Observation: No feasible static partitioning strategy exists.** Under pFP, tasks are statically assigned to cores (e.g., as done in [18, 29]) and shared-cache isolation is achieved among tasks on different cores via static cache partitioning. However, this static approach cannot schedule the example workload.

---

12Recall that the maximum number of partitions a task can have is 12.

13The nFPca-favor tasksets favor nFPca in comparison to gFPca.

14We observed similar results when varying the number of tasks.

15The upper bound is to account for potential sources of interference, such as TLB overhead, and variable actual program execution time.
Specifically, since $\tau_1$ requires all of 16 cache partitions to meet its deadline, if we allocate less than 16 partitions to its core, then it will miss its deadline. If we allocate all 16 cache partitions to $\tau_1$’s core, then either (i) some lower-priority task will have zero cache partition (if it is assigned to a different core) and will miss its deadline, or (ii) all tasks must be packed onto the same core as $\tau_1$’s, in which case the taskset is unschedulable (since the core utilization is more than 1). In other words, no partitioning strategy exists for the workload.

**Experiment**: The reference taskset illustrates the scenario where the high-priority task has a very high density (ratio of WCET to deadline) and thus is extremely sensitive to interference. To investigate the impact of task density on the performance of the algorithms, we varied the density of $\tau_1$ from 1 to 0.1 by increasing its deadline (while keeping all the other parameters unchanged), which produced 10 tasksets. The number of cache partitions were assigned for gFPca and nFPca as above ($A_1 = 16$ and $A_i = 4$, with $i > 1$). Although our analysis shows that no feasible partitioning strategy exists for pFP, for validation we evenly distributed four low-priority tasks and 16 cache partitions to the four cores, and assigned $\tau_1$ to any of the four cores. We ran each generated taskset for one minute under each of the four schedulers (gFPca, nFPca, gFP, pFP) schedulers, collected their scheduling traces, and derived the observed schedulability under each scheduler.

**TABLE 1: Impact of task density on schedulability.**

<table>
<thead>
<tr>
<th>Density</th>
<th>0.8</th>
<th>0.7</th>
<th>0.6</th>
<th>0.5</th>
<th>0.4</th>
<th>0.3</th>
<th>0.2</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>gFPca</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>gFP</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nFPca</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>pFP</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Results.** Table 1 shows the observed schedulability of each taskset under each scheduler. The results show that the gFPca scheduler performed best: it was able to schedule all tasksets. The gFP scheduler performed well when the high-priority task’s density is low; however, as the task’s deadline becomes tighter, its tolerance to cache interference from other tasks is decreased, and thus it began to miss its deadline. The results also show that the nFPca scheduler performed very poorly: it was able to schedule only one taskset; we attribute this to its poor utilization of cache and CPU resources due to its non-preemptive nature. As predicted in our analysis, the pFP scheduler could not schedule any tasksets.

**X. CONCLUSION**

We have presented the design, implementation and analysis of gFPca, a cache-aware global preemptive fixed-priority scheduling algorithm with dynamic cache allocation. Our implementation has reasonable run-time overhead, and our overhead analysis integrates several novel ideas that enable highly accurate analysis results. Our numerical evaluation, using overhead data from real measurements on our implementation, shows that gFP improves schedulability substantially compared to the cache-agnostic gFP, and it outperforms the existing cache-aware nFPca in most cases. Through our empirical evaluation, we illustrated the applicability and benefits of gFPca. For future work, we plan to enhance both gFPca and our implementation to improve its efficiency and performance.

**REFERENCES**

**Appendix A: gFPca Run-time Overhead**

We used the feather-trace tool to measure the overheads, as in earlier LITMUS\textsuperscript{RT} based studies (e.g., [11, 12]). Since the tool uses the timestamp counter to track the start and finish time of an event in cycles, we first validated that the timestamp counter on our board has a constant speed (necessary for precise conversion from cycles to nanoseconds). Since the timestamp counter on each core of the board is not synchronized, we also modified the tool to use the system-wide monotonically-increasing timer (in nanosecond) to trace the IPI delay.

We randomly generated periodic tasksets of size ranging between 50 to 450 tasks, with a step of 50. We generated 10 tasksets per taskset size (i.e., 90 tasksets in total) under each scheduler. Under each scheduler, we traced each taskset for 30 seconds, and measured all size types of overhead: release overhead, release latency, scheduling overhead, context switch overhead, IPI delay, and tick overhead (as defined in [1]). We removed the outliers using the method in [11] and computed the worst-case and average-case overheads.

<table>
<thead>
<tr>
<th>Taskset size: 50</th>
<th>Taskset size: 450</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( gEDF )</td>
</tr>
<tr>
<td>Release</td>
<td>5.72</td>
</tr>
<tr>
<td>Sched</td>
<td>8.64</td>
</tr>
<tr>
<td>CXS</td>
<td>4.25</td>
</tr>
<tr>
<td>IPI</td>
<td>4.06</td>
</tr>
</tbody>
</table>

**TABLE 2: Average overhead (\( \mu s \)) under different schedulers with cache-read workload.**

Table 2 shows the average overheads for taskset size of 50 and 450 under the \( gFPca \) and nFPca schedulers, as well as the existing \( gEDF \) scheduler in LITMUS\textsuperscript{RT} for comparison. The results show that the release, scheduling, and IPI delay overheads of the \( gFPca \) and nFPca schedulers are similar to that of \( gEDF \). However, \( gFPca \) and nFPca have a larger context switch overhead than \( gEDF \) does, which is expected because they may need to flush cache partitions during a context switch, as described in the implementation description. The \( gFPca \) scheduler incurs higher worst-case overheads than the \( gEDF \) scheduler, which is not surprising because the scheduling algorithm \( gFPca \) has a higher complexity than \( gEDF \). All measured overhead values can be found in [31].

**Appendix B: Overhead Analysis Challenges**

Existing overhead accounting approaches [3, 12, 22, 26, 32] typically work as follows:

- first, analyze for each task \( \tau_i \) either (a) the maximum cache overhead, \( \theta_i \), that \( \tau_i \) causes to its lower-priority tasks, or (b) the maximum cost of one cache overhead, \( \Delta_i \), that \( \tau_i \) incurs upon resuming from a preemption;
- then, incorporate the overhead into the analysis by inflating the tasks’ WCETs based on the obtained \( \theta_i \) or \( \Delta_i \).

It seems intuitive at first to apply the same approach for \( gFPca \); unfortunately, a naïve computation of \( \theta_i \) or WCET inflation based on \( \Delta_i \) can lead to unsafe analysis results for \( gFPca \).

(Note: these apply only to \( gFPca \), not \( gFP \).) We will show this using an example; see [31] for more examples.

**Naïve WCET inflation based on \( \theta_i \).** We first compute \( \theta_i \) for each task \( \tau_i \) \((i < n)\) and then inflate \( \tau_i \)’s WCET by the overhead \( \theta_i \). For this, we extend the method used in the uniprocessor setting [28]. Specifically, when a higher-priority task \( \tau_i \) preempts \( \tau_k \) on a uniprocessor, the cache lines that \( \tau_i \) may evict from the cache must be the cache lines it can access, i.e., its ECBs (c.f. Section III). Let \( BRT \) be the maximum latency of reloading one cache line and \( ECB_i \) be the ECBs of \( \tau_i \). Thus, the private-cache overhead caused by \( \tau_i \) is bounded by [28]: \( \theta_i^{\text{ini}} = BRT \times |ECB_i| \). It seems intuitive to apply the same idea to \( gFPca \) by using the ECPs of \( \tau_i \), since the partitions that \( \tau_i \) evicts should be the partitions it can access.

Recall that \( PRT \) is the latency of reloading one cache partition and \( ECP_i \) is \( \tau_i \)’s ECPs. Then, the cache overhead caused by \( \tau_i \) is bounded by \( \theta_i = PRT \times |ECP_i| \). However, this bound is unsafe when applied to \( gFPca \), as shown in the example below.

**Counter Example 1.** Consider a taskset \( \tau = \{ \tau_1, \tau_2, \tau_3 \} \), with \( \tau_1 = (12, 2, 10, 2) \), \( \tau_2 = (12, 4, 11, 7) \), \( \tau_3 = (12, 6, 12, 5) \), and priority order \( \tau_1 > \tau_2 > \tau_3 \). Suppose \( \tau \) is scheduled using \( gFPca \) on a dual-core platform with 8 cache partitions, and \( \tau_1, \tau_2, \) and \( \tau_3 \) are released at time 4, 2, and 0, respectively.

Suppose \( PRT = 0.2 \). Then, as illustrated in Fig. 9(a), \( \tau_3 \) finishes at \( t = 12.4 \) and thus misses its deadline.

However, from \( \theta_i = PRT \times |ECP_i| \), we obtain \( \theta_1 = 0.4 \) and \( \theta_2 = 1.4 \). If we inflate \( \tau_1 \) and \( \tau_2 \) with \( \theta_1 \) and \( \theta_2 \), respectively, then their inflated WCETs are \( e_i^1 = 2.4 \) and \( e_i^2 = 5.4 \). As illustrated in Fig. 9(b), this leads to \( \tau_3 \) finishing at \( t = 11.4 \) and meeting its deadline. Clearly, the inflated WCETs are insufficient to account for the actual overhead \( \tau_3 \) experiences.

Alternatively, if we inflate the WCET of each low-priority task (\( \tau_2 \) and \( \tau_3 \)) with the total cache overhead caused by all of its higher-priority tasks, the inflated WCET of each task will be: \( e_i^1 = 2 \), \( e_i^2 = e_3^2 + |p_2/p_1| \times \theta_1 = 4.4 \); and \( e_i^3 = e_3 + |p_3/p_1| \times \theta_1 + |p_3/p_2| \times \theta_2 = 7.8 \). Then \( \tau_3 \) would finish at \( t = 12.2 \) which is earlier than its actual finish time.

As Fig. 9(a) illustrates, under \( gFPca \) the cache effects \( \tau_i \) has on a lower-priority task \( \tau_k \) comes from not only direct preemption (i.e., when \( \tau_i \) is released and preempts \( \tau_k \)) but also indirect preemption: when \( \tau_i \) is released, it is possible that \( \tau_i \) and \( \tau_k \) are scheduled to run whereas an intermediate-priority \( \tau_j \) \((i < j < k)\) is blocked due to insufficient cache; when \( \tau_i \) finishes, \( \tau_k \) is preempted by \( \tau_i \) because there is now sufficient cache for \( \tau_i \) to execute. Therefore, the number of cache partitions of \( \tau_k \) that are evicted can be as large as \(|ECP_j| + |ECP_k|\) (which is more than \(|ECP_j|\)).

**Appendix C: Baseline Analysis**

We describe the overhead-aware analysis for \( gFPca \) that was used as the baseline in our numerical evaluation. This baseline method performs WCET inflation based on the cache overhead \( \Delta_i \) that each task \( \tau_i \) incurs upon resuming from a preemption. However, instead of inflating the WCET of each high-priority task with the maximum of one cache overhead of its lower-priority tasks (which is unsafe), it inflates the WCET of each \( \tau_i \) with its total cache overhead (i.e., the overhead it experiences during the entire execution of a job).

**Computing the total overhead of \( \tau_i \):** The cache overhead that \( \tau_i \) experiences when it resumes from a preemption is upper bounded by \( \Delta_i \leq PRT \times |UCP_i| \). Since a cache partition of \( \tau_i \) may be evicted from the cache only when another task \( \tau_j \) uses
Inflating WCETs of high-priority tasks

Lemma 11. The cache overhead a task \(\tau_i\) experiences when it resumes from one preemption is upper bounded by \(\Delta \leq PRT \times \left| UCP_i \cap \bigcup_{j \neq i} ECP_j \right| \leq PRT \times \min\{\left| UCP_i \right|, \sum_{j \neq i} \left| ECP_j \right|\} \).

To bound the total cache overhead of \(\tau_i\), we next derive the maximum number of times that \(\tau_i\) resumes (i.e., number of resumption events of \(\tau_i\)) in each job’s execution.

Lemma 12. A task \(\tau_i\) resumes only when one of the following two events happens: a higher-priority task of \(\tau_i\) finishes its execution, or a higher-priority task of \(\tau_i\) releases a new job.

Lemma 13. The maximum number of task-resumption events of \(\tau_i\) during each period is at most \(N_{S_i} = \sum_{j < i} 2^\left\lfloor \frac{d_j}{p_j} \right\rfloor + 2\).

Since \(\tau_i\) only incurs (extrinsic) cache overhead whenever it resumes, the total overhead of \(\tau_i\) is therefore at most \(N_{S_i} \times \Delta_i\).

Overhead-aware analysis: Since the total overhead of \(\tau_i\) is at most \(N_{S_i} \times \Delta_i\), the WCET of \(\tau_i\) in the presence of cache overhead is at most \(e_i' = e_i + N_{S_i} \times \Delta_i\). As a result, the overhead-aware analysis can be established by applying the overhead-free analysis on the inflated workload.

Appendix D: Extension to Other Overhead Types

Real-time tasks typically experience six major sources of overhead [10]: release, scheduling, context-switching, IPI overhead, cache related preemption and migration (CRPMD), and tick overheads. We specify the cost of each of these six overheads as \(\Delta^{rel}, \Delta^{sched}, \Delta^{ctx}, \Delta^{IPI}, \Delta^{CRPMD}\), and \(\Delta^{tick}\). Since the tick overhead is quite small (<11\,\mu s for 450 tasks on our board) and does not involve any scheduling-related logic under all three (event-driven) schedulers (gFPca, nFPca, and gFP), we exclude it from the analysis and focus on the other five types of overhead. (Our analysis does not consider blocking overhead.) We first analyze the overhead when a task executes alone, and then account for all types of preemption-related overhead. We then perform WCET inflation, and apply the overhead-free schedulability analysis on the inflated taskset.

Overhead accounting when a task executes alone. We observe that a task \(\tau_k\) always incurs one release overhead, one IPI delay overhead, one scheduling overhead, and one context switch overhead, when it executes alone in the system under any of the three schedulers. Therefore, the execution time \(e_{k}' = e_k + \Delta^{rel} + \Delta^{IPI} + \Delta^{sched} + \Delta^{ctx}\) is a safe bound on the execution time \(e_k\) of \(\tau_k\) in the presence of the overhead when the task executes alone.

Overhead accounting under gFPca. Fig. 10 illustrates the preemption-related overhead under gFPca. We observe that at the same cache partition, we can tighten \(\Delta_i\) by considering the cache partitions used by other tasks:

\[
\delta_k = \left[(\alpha_k \times e_k) / \text{hit}\_\text{latency}\right] \times (\text{miss}\_\text{latency} - \text{hit}\_\text{latency})
\]

Therefore, the inflated execution time of \(\tau_k\) that accounts for five types of overhead is bounded by

\[
e_{k}' = e_k + \Delta^{rel} + \Delta^{IPI} + \Delta^{sched} + \Delta^{ctx} + \delta_k
\]
Overhead-aware analysis. For each scheduler (i.e., gFPca, nFPca and gFP), the overhead-aware analysis can now be achieved by applying its overhead-free analysis to the inflated taskset with the inflated WCET computed above.