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  
IEEEExplore page  

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

Meng Xu  Linh Thi Xuan Phan  Hyon-Young Choi  Insup Lee

University of Pennsylvania

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 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 task sets 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.

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 [15] or way partitioning [20]. 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 [6, 16, 22], where cache partitions and tasks are assigned to specific cores offline. While this approach makes the multicore analysis simpler, it can significantly under-utilize 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. [13] 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 [13] 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 \( \tau_1 \) completes its execution (which enables \( \tau_2 \) to resume its execution and thus preempt \( \tau_3 \)). 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 \( \tau_3 \) to continue).

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 conditions 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 this new overhead accounting approach is highly accurate.

In summary, we make the following specific contributions:

- the \( gFPca \) scheduling algorithm (Section IV);
- an implementation of \( gFPca \), as well as \( nFPca \) and \( gFP \), on an existing multicore hardware platform. (Section V);
- an overhead-free analysis for \( gFPca \) (Section VI); and
- 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, 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 [12, 14, 19]. 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. [3, 11] 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, our analysis formally computes the cache overhead due to shared cache interference, rather than assuming that this overhead is given.

Schedulability analysis methods that consider the cache preemption or migration overhead have been proposed [4, 10, 17, 23, 26, 28]. 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 new approach.

There also exist shared cache management techniques for multicore platforms; for instance, Ward et al. [25] 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 [6, 16, 21, 22]. Existing work typically considers a partitioned scheduling approach, which statically allocates tasks to different shared-cache partitions and then to different cores. This approach cannot always be feasibly applied in practice; 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 the cache content of each other, which can result in deadline misses. (See Section IX for an example.) 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. [13], 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 [13] 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 [13] 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 [20]. The latency of 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. As a first step, this paper focuses on the shared-cache interference and considers only data caches; we assume that the effects of other resource interferences, such as that of private caches and memory bus, are negligible or have been included in the tasks’ WCETs.

The system consists of a set of independent explicit-deadline sporadic tasks, \( \tau = \{ \tau_1, ..., \tau_n \} \). Each task \( \tau_i \) is defined by
\( \tau = (p_i, e_i, d_i, A_i) \), 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 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 \( e_i \). (In our numerical evaluation, \( A_i \) was chosen to be the smallest number of cache partitions that leads to the minimum WCET for \( \tau_i \).) In addition, 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 \( \tau_i \in \tau \), where \( A \) is the total number of partitions of the shared cache. Each task has a fixed and unique priority; without loss of generality, we assume that 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 miss; otherwise, it is an extrinsic cache miss [5]. The overhead (of reloading the evicted cache content) due to intrinsic cache misses of a task can typically be analyzed statically 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; for example, 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. In this paper, we consider only the shared cache overhead and defer the incorporation of the private cache overhead to future work.

**ECP and UCP.** We say that a task accesses a partition if it accesses any line(s) within that partition. We define an Evicting Cache Partition (ECP) of a task to be a cache partition that the task can access, and we denote by \( \text{ECP}_k \) the set of ECPs of \( \tau_i \) during an uninterrupted execution interval of \( \tau_i \). Note that \( \text{ECP}_k \) varies across different continuous execution intervals of \( \tau_i \), but \( |\text{ECP}_k| \leq A_i \) by definition. In addition, we define a Useful Cache Partition (UCP) of \( \tau_i \) to be a cache partition that \( \tau_i \) accesses at some time point and later accesses again as cache hit, when \( \tau_i \) executes alone in the system. The set of UCPs of \( \tau_i \) is denoted by \( \text{UCP}_k \); by definition, \( \text{UCP}_k \subseteq \text{ECP}_k \).

### IV. gFPca Scheduling Algorithm

We now present the gFPca algorithm. Like gFP, gFPca also schedules tasks based on their 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 \( \tau_i \):

1. **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.

2. **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 for \( \tau_i \), 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. (When more than \( A_i \) partitions are found, gFPca gives preference to the ones that still hold the cache content of the task \( \tau_i \).) Otherwise, gFPca will move to the next task and repeat the process from Step 1. 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. Due to space limitation, we omit the details here.

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 analysis considers the costs of mapping the memory accesses and reloading the memory content into the cache. In our implementation, reassigned partitions can be done by simply resetting the registers that control the cache partitions (without the need to copy memory pages), which takes only about a few cycles; therefore, we consider the overhead of reassigning partitions as part of the context switch overhead in our analysis.

### V. IMPLEMENTATION

We implemented gFPca within LITMUSRT on the Freescale I.MX6 quad-core evaluation board, which supports way partitioning through the PL310 cache controller. For comparison, we also implemented the existing non-preemptive nFPca in [13] 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 [20, 25]). 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 $R_i$, where a bit $q$ in $R_i$ is one if the cache allocation cannot happen in the cache way $q$ for the memory access from the core $P_i$, and zero otherwise. (To be precise, each core has two separate registers for instruction and data access, but we focus on data access in this paper.)

**Challenge.** To reserve the set of cache partitions $S_k$ (represented as a bitmask) for a task $t_k$ on a core, we set the lockdown register of the core to be the bitwise complement of $S_k$. However, this alone cannot guarantee that $t_k$ will not access cache partitions outside $S_k$, because the LbM cannot control where the cache lookup (i.e., cache hit) occurs. As a result, tasks running concurrently on different cores may still access each other’s cache partitions, even if the register is set.

**Approach:** Recall that the actual cache partitions allocated to a task varies from one preemption point to the next (even within the same job of the task). One way to address the above challenge is to flush the partitions allocated to each task $t_k$ when it completes a job or is preempted [25]. However, this approach prevents a task from reusing its content in the cache when possible: if a partition reserved for $t_k$ has not been used by any other task when $t_k$ resumes or releases a new job, then $t_k$ should be able to reuse the content inside that partition; this would not be possible if we had flushed the task’s partitions when it was preempted or finished its previous job.

Since the cost of flushing a cache way is relatively expensive compared to other scheduler-related overheads, we minimized cache flushes through selective flushing. The idea is to select from the reserved partitions of $t_k$ all the partitions that may hold the content of other tasks, and then flush the selected partitions when $t_k$ resumes or releases a new job.

To flush a cache partition, we leveraged the hardware cache maintenance operations to clean and invalidate the specific cache ways that need to be flushed. (This is different from the approach in [25], which loads pages to the cache partitions to evict the previous content from the cache.) Our approach guarantees cache isolation among concurrently running tasks (since no task can use the reserved cache partitions of another task), and helps to minimize the cache management overhead (since a task may use the previously reserved – rather than currently reserved – partitions until they are reserved and flushed by another task). Note that when the cache content of a task $t_k$ is flushed from its previously reserved partitions (by another task), then $t_k$ may need to reload its content to its current reserved partitions; we account for such overhead in our analysis.

**B. Scheduling architecture**

Fig. 1 shows a high-level overview of the scheduling architecture for gFPca. Our implementation extended various components in LITMUS$^{RT}$ to incorporate gFPca’s cache management and scheduling behavior. Most notable extensions include: (1) **RT Task**: We extended the 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) **RT-Context**: We extended the cpu_entry data structure, which holds the real-time context of a core, with a new field called preempting to indicate whether the core is preempted via cache. (3) **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 gFPca scheduler**: The steps in Fig. 1 illustrates 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** Execute the 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 $t_i$ and $t_j$, respectively) and schedule the ready task (say $t_k$) 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 $t_i$ and $t_j$ and then marking the partitions that will be reserved for $t_k$. In addition, it updates P0’s linked task (i.e., the real-time task to execute next) to be $t_k$, P2’s linked task to be NULL and P2’s 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 \( \tau_k \), and P2 to preempt \( \tau_j \)).

When a core receives the above IPI, the scheduler on that 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. We assume that non-real-time tasks do not interfere with the real-time tasks.)

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 CP\text{to}Task-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 gFP\text{ca} and nFP\text{ca} schedulers, and the existing gEDF scheduler in LITMUS\textsuperscript{RT}. We observed that both gFP\text{ca} and nFP\text{ca} 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 gFP\text{ca} scheduler incurs higher worst-case overheads than the gEDF scheduler, which is expected because the gFP\text{ca} algorithm has a higher complexity. (Additional details are described in Appendix A.)

In the coming sections, we present the schedulability analysis of gFP\text{ca}, first assuming the absence of overhead and then considering all types of overhead. As the analysis of the cache-related preemption and migration delay (CRPMD) overhead is most challenging, we focus on the analysis of the CRPMD overhead in the main context and present the extension to the remaining types of overhead in Appendix C. Note that our evaluation considered all these overheads. Due to space constraints, we omit most of the proofs here, but they are available in [27].

VI. OVERHEAD-FREE ANALYSIS

The overhead-free schedulability analysis of gFP\text{ca} can be established using a similar idea as that of nFP\text{ca} [13]. As usual, the processor demand of a task \( \tau_i \) in any interval \([a,b]\) is the amount of processing time required by \( \tau_i \) in \([a,b]\) that has to complete at or before \( b \). When task \( \tau_k \) is scheduled under gFP\text{ca}, \( \tau_k \) 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 \( \tau_k \) and the following jobs of \( \tau_k \) execute as early as possible, as illustrated in Fig. 2. Hence, the worst-case demand of \( \tau_k \) in a period of \( \tau_k \) is given by [7]:

\[
W_k = NJ_k e_i + \min\{d_k + d_l - e_i - NJ_k p_i, e_i\},
\]

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

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 (i.e., idle or executing a lower-priority task) and at least \( A - A_k + 1 \) cache partitions are assigned to \( \tau_k \)'s higher-priority tasks.

Consequently, the workload of \( \tau_k \) in a period of \( \tau_k \) consists of two types: (1) CPU-interference workload, \( \alpha_k \), which is the workload of \( \tau_k \) when it executes in the CPU-busy interval of \( \tau_k \); and (2) cache-interference workload, \( \beta_k \), which is the workload of \( \tau_k \) when it executes in the cache-busy interval of \( \tau_k \). Since \( \tau_k \) cannot execute when its higher-priority tasks collaboratively keep the CPU busy, and because the system has \( M \) cores, the length of the CPU-busy interval of \( \tau_k \) is bounded by \( \frac{1}{M} \sum_{\tau_i < k} \alpha_i \). Because each higher-priority task executes \( \beta_k \) time units with \( A_k \) cache partitions occupied, and because higher-priority tasks only need to occupy \( A - A_k + 1 \) cache partitions to prevent \( \tau_k \) from execution, the combined cache resources (i.e., the number of partitions occupied in an interval multiplied by the interval length) that need to be used by all other tasks to block \( \tau_k \) from execution during \( \tau_k \)'s cache-busy interval is bounded above by \( \sum_{\tau_i < k} \min\{A_i, A - A_k + 1\} \beta_i \). Therefore, the length of the cache-busy interval of \( \tau_k \) is bounded above by \( \sum_{\tau_i < k} \min\{A_i, A - A_k + 1\} \beta_i \). Since the length of the busy interval of \( \tau_k \) is no more than the sum of the length of the CPU-busy interval and the length of the cache-busy interval, it is bounded above by:

\[
\sum_{\tau_i < k} \left( \frac{1}{M} \alpha_k + \min\{A_i, A - A_k + 1\} \beta_i \right).
\]

Further, in each period of \( \tau_k \), the CPU/cache-interference workload of a higher-priority task \( \tau_i \) must satisfy the following constraints: (1) the combination of the CPU-interference workload and cache-interference workload of \( \tau_i \) cannot exceed the workload of \( \tau_k \), i.e., \( \alpha_k + \beta_i \leq W_k \); and (2) the CPU/cache-interference workload of \( \tau_i \) should be no more than the length of the CPU/cache-busy interval of \( \tau_i \), i.e., \( \alpha_k \leq \sum_{\tau_i < k} \frac{1}{M} \alpha_i \) and \( \beta_i \leq \sum_{\tau_i < k} \min\{A_i, A - A_k + 1\} \beta_i \).

Based on the above discussion, we obtain the following:

**Lemma 1.** The maximum length \( B_k \) of the busy interval of \( \tau_k \) is bounded by \( B_k \), where \( B_k \) is the optimal solution of the following Linear Programming (LP) problem:

\[
\begin{align*}
\text{maximize} & \quad \sum_{\tau_i < k} \left( \frac{1}{M} \alpha_i + \min\{A_i, A - A_k + 1\} \beta_i \right) \\
\text{subject to} & \quad \alpha_i + \beta_i \leq W_k, \quad \forall i < k \\
& \quad \alpha_i \leq \sum_{\tau_i < k} \frac{1}{M} \alpha_i \\
& \quad \beta_i \leq \sum_{\tau_i < k} \min\{A_i, A - A_k + 1\} \beta_i 
\end{align*}
\]

**Proof.** The lemma holds by construction as discussed above.

\[\square\]

The next theorem follows as a result of Lemma 1.
Theorem 2. A taskset $\tau$ is schedulable under the gFPca algorithm if each task $\tau_k$ in $\tau$ satisfies $B_k \leq d_k - e_k$.

Theorem 3. Given a taskset $\tau = \{\tau_1, ..., \tau_n\}$, where $\tau_i = (p_i, e_i, d_i, A_i)$ for all $1 \leq i \leq n$. Let $\tau = \{\tau_1, ..., \tau_n\}$ be any task set with $\tau_i = (p_i, e_i, d_i, A_i)$ and $e_i \leq e_i$ for all $1 \leq i \leq n$. Then, $\tau$ is schedulable under the gFPca algorithm if $\tau$ satisfies the gFPca schedulability conditions given by Theorem 2.

Proof. We will show that if $\tau$ is unschedulable under gFPca, then $\tau$ will be deemed unschedulable under Theorem 2. Indeed, if $\tau$ is unschedulable under gFPca, then there exists a task $\tau_i \in \tau$ that misses its deadline. Let $B_k$ be the maximum length of the busy interval of $\tau_i$. Then, $B_k \geq B_k$ due to Lemma 1. Since $\tau_i$ misses its deadline, $B_k \geq d_k - e_k$. Combining this with $e_i \geq e_i$ and $B_k \geq B_k$, we obtain $B_k \geq d_k - e_k$. Thus, the taskset $\tau_i$ is deemed unschedulable by Theorem 2. \hfill \Box

VII. OVERHEAD-AWARE ANALYSIS

Insight. We observe that under gFPca, the cache effects $\tau_i$ has on a lower-priority task $\tau_k$ comes from not only direct preemption (i.e., $\tau_j$ 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.e., $d_k < e_k$) is blocked due to insufficient cache for it; when $\tau_i$ finishes, $\tau_i$ is preempted by $\tau_j$ because there is now sufficient cache for $\tau_i$ to execute. Due to this behavior, existing approaches, such as [24], cannot be applied. (A detailed discussion of the challenges is available in our technical report [27].)

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 $\tau_i$ via CPU or cache resource. Finally, we analyze the total overhead of $\tau_i$ that is caused by the cache-related events of other tasks and include it into $\tau_i$’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 $\tau_i$ occurs when the CPU or cache resource allocated to $\tau_i$ 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 $\tau_i$ with the task-release or task-resumption event may take the CPU and/or cache resource from $\tau_i$, thus leading to a task-preemption event of $\tau_i$. Similarly, because running jobs may stop at task-preemption and task-finish events, and the released CPU or cache resource may be allocated to $\tau_i$, both task-preemption and task-finish events of $\tau_i$ may lead to a task-resumption event of $\tau_i$. 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 4. Task-release events and task-finish events are the source events that cause overhead in a system.

Based on Lemma 4, if we can compute a bound on the overhead that each task-release event and each task-finish event of a higher-priority task cause to a lower-priority task, then we can safely account for the total overhead of $\tau_i$. To derive this bound, we will analyze the set of tasks that can preempt $\tau_i$ based on the necessary conditions of task-preemption events, which we now establish.

B. Conditions of task-preemption events

The overhead that a task $\tau_i$ 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 $\tau_i$ via either CPU and/or cache resources; however, no task-preemption event of $\tau_i$ 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 5. 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 6. If $\tau_i$ preempts $\tau_k$’s cache resource at $t$, 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 $A - A_i$, where $A$ is the number of cache partitions of the cache.

Let $\rho_i$ and $k_i$ be the maximum sets of tasks that may preempt $\tau_i$ via CPU and cache resources, respectively. Due to the above lemmas, we have:

$$\rho_i = \{ \tau_i \mid i < k \text{ and } \sum_{j < k} 1 \geq M \} \tag{2}$$

$$k_i = \{ \tau_i \mid i < k \text{ and } \sum_{j < k} A_j > A \} \tag{3}$$

As a result, the set of tasks that may preempt $\tau_i$ via either CPU or cache or both resources is $\rho_i \cup k_i$. 

![Fig. 3: Causal relations of task events.](image-url)
C. Overhead caused by a task-release event

Based on the established conditions of a task-preemption event of $τ_k$, we can analyze the overhead of $τ_k$ that is caused by one task-release event of a higher-priority task $τ_i$.

Observe that when $τ_i$ releases a job at time $t_r$, the cache partitions $τ_i$ may access and pollute are in ECP$_i$. If $τ_k$ is preempted at the task-release event of $τ_i$, $τ_i$ can directly evict all cache partitions in ECP$_i$ that $τ_k$ may use in the worst case.

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

**Lemma 7.** When a release-event of $τ_i$ occurs, if $τ_k$ is preempted but a lower-priority task $τ_j$ ($k < l$) either resumes from a preemption or releases a new job and this job is executed, then the number of cache partitions of $τ_i$ must be less than that of $τ_k$, i.e., $A_j < A_k$.

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

$$φ_{l,k}^f = \begin{cases} 
\text{UCP}_k \cap \left( \text{ECP}_l \cup \bigcup_{k < l, A_j < A_l} \text{ECP}_j \right), & \text{if } τ_i \in K_k \\
\text{UCP}_k \cap \left( \bigcup_{k < l, A_j < A_l} \text{ECP}_j \right), & \text{if } τ_i \not\in K_k \land τ_i \in \rho_k \\
\emptyset, & \text{if } τ_i \not\in \{K_k \cup \rho_k\}
\end{cases} \quad (4)$$

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,

$$|φ_{l,k}^f| \leq \min\{|\text{UCP}_k|, \big|\bigcup_{k < l, A_j < A_l} \text{ECP}_j\|\}, \text{ if } τ_i \not\in K_k \land τ_i \in \rho_k \\
0, \text{ if } τ_i \not\in \{K_k \cup \rho_k\}
\quad (5)$$

Denote by $Δ_{l,k}^f$ the overhead of $τ_k$ that is caused by a task-release event of $τ_i$, where $i < k$. Then,

$$Δ_{l,k}^f \leq \text{PRT} \cdot |φ_{l,k}^f| \quad (6)$$

D. Overhead caused by a task-finish event

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

**Case 1** $τ_k$ is not running at $t_f$: If $τ_k$ finishes before or at $t_f$, then clearly the task-finish event causes no overhead to $τ_i$.

If it has not finished its execution at $t_f$, this task-finish event also does not bring any overhead to $τ_k$, because even though $τ_i$ might have polluted $τ_k$’s cache before $t_f$, the pollution is caused by other task-release or task-finish events of $τ_i$ and should be accounted in the cost of those events.

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

**Lemma 8.** If a task $τ_j$, where $i < j < k$, resumes and preempts $τ_k$ at a task-finish event of $τ_i$, then $A_j > A_k$.

**Lemma 9.** There exists at most one task $τ_j$ with $i < j < k$ that can resume and preempt $τ_k$ at a task-finish event of $τ_i$.

In addition, when $τ_k$ is preempted, lower-priority tasks of $τ_k$ may also resume or release new jobs and these jobs are executed, and thus they may pollute the cache partitions of $τ_k$. According to Lemma 7, only lower-priority tasks $τ_j$ with $k < l$ and $A_j < A_k$ may pollute $τ_k$’s cache partitions while $τ_k$ is being preempted. When a task $τ_j$ ($i < j < k$) resumes and preempts $τ_k$ at the occurrence of the task-finish event of $τ_i$, the set of useful cache partitions of $τ_k$ that may be polluted, denoted by $φ_{l,k}^f$, is the same with the set of useful cache partition of $τ_k$ that may be polluted at the task-release event of $τ_i$. Therefore,

$$φ_{l,k}^f = φ_{l,k}^r \text{ and the size of } φ_{l,k}^f \text{ is } |φ_{l,k}^f| = |φ_{l,k}^r|.$$

Let $Δ_{l,k}^f$ denote the overhead of $τ_k$ that is caused by a task-finish event of $τ_i$, where $i < k$. Because any task $τ_j$ ($i < j < k$ and $A_j < A_k$) may resume and preempt $τ_k$ at the task-finish event of $τ_i$, we obtain

$$Δ_{l,k}^f \leq \max_{i < j < k, A_j < A_k} \text{PRT} \cdot |φ_{l,k}^f| \quad (7)$$

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 $τ_k$ causes to a lower-priority task $τ_i$. To account for the overall overhead $τ_k$ experiences, we need to compute the number of task-release and task-finish events of higher-priority tasks in each period of $τ_k$.

Since each job of a task has a 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 $τ_k$ is $\sum_{i < k} 2\left[\frac{A_i}{P_i}\right] + 2$. While this bound is safe, it is not tight because not every task-release event or task-finish event of each job of higher-priority tasks can cause overhead to $τ_k$, as stated by Lemma 10.

**Lemma 10.** If a task $τ_k$ is preempted at the release time $t_r$ and again at the finish time $t_f$ of the same job of a higher-priority task $τ_i$, then $τ_k$ must have been restarted at some time $t_r$ during the interval $(t_1, t_2)$ when some other higher-priority task $τ_j$ ($j < k$) releases or finishes.

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 $τ_k$ that is caused by each job of its higher-priority tasks in a period of $τ_k$, 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 max \{\( \Delta_i^r, \Delta_i^f \)\}. In contrast, if both the task-release and task-finish events of the same job of \( \tau_i \) may cause overhead to \( \tau_k \), the maximum overhead of \( \tau_k \) that is caused by each job of \( \tau_i \) is the total overhead caused by the task-release and task-finish events of the job \( \tau_i \) minus the minimal overhead caused by the task-release event or the task-finish event of a high-priority task \( \tau_j \) (\( j < k \) and \( j \neq i \)), i.e., \( \Delta_i^r + \Delta_i^f - \min_{j < k, j \neq i} \{\Delta_j^r, \Delta_j^f\} \). Hence, the overhead of \( \tau_k \) that is caused by one job of a higher-priority task \( \tau_i \) is bounded by

\[
\delta_i^k \leq \max\{\Delta_i^r, \Delta_i^f, \Delta_i^r + \Delta_i^f - \min_{j < k, j \neq i} \{\Delta_j^r, \Delta_j^f\}\}.
\]

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 \( N_t^k \leq \frac{d_k}{p_k} \). Since the finish event of the carry-in job of \( \tau_i \) and the release event of the carry-out job of \( \tau_i \) in a period of \( \tau_i \) 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

\[
\delta^k = \sum_{i=1}^{k-1} \delta_i^k \cdot N_t^k + \Delta_i^r + \Delta_i^f (8)
\]

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 11.** 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 2, 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 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. Details are in Appendix B.

**Types of overhead.** Besides CRPMD overhead, our evaluation considered four other types: release, scheduling, IPI, and context switch. For this, we extended the analysis in Section VII, in [13], and in [7], 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 the uniform distribution within the range [0.5, 0.9] as used in [26, 28]. The number of ECPs of a task was uniformly distributed in [1, 8] by default. The number of UCPs was set equal to the number of ECPs (i.e., we considered the conservative case of our theory, where the UCPs and ECPs of a task are the same).

**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 16 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 = 0.19ms.

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 [27]).

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

We generated 4000 tasksets with taskset utilization ranging from 0.1 to 4, with a step of 0.1. For each taskset utilization, there were 100 independently generated tasksets; the task utilizations were uniformly distributed in [0.5, 0.9]; the task periods were uniformly distributed in [10, 40]ms. (These parameters followed existing work such as [26, 28]). Fig. 4 shows 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 2.5, the baseline analysis claimed that only 5% of the tasksets are schedulable, even though 64% 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 4000 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] (this was chosen based on [18]). 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 hits_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 nFPca 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 in Fig. 5 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 utilizes the system’s resources better.

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. Recall that the maximum number of partitions a task can have is 12. 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 tasksets (i.e., tasksets that favor nFPca in comparison to gFPca), which have $|A_i| = \left\lceil \frac{\max_{\text{period}} - \min_{\text{period}}}{\text{max_period} - \text{min_period}} \times 12 \right\rceil$ for each $\tau_i$, and (2) the so-called nFPca-oppose tasksets, in which $|A_i| = 12 - \left\lceil \frac{\max_{\text{period}} - \min_{\text{period}}}{\text{max_period} - \text{min_period}} \times 12 \right\rceil$ 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 and 7. 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 also observe that the performance improvement that gFPca achieves over nFPca increases as the tasksets move from the nFPca-favor to the nFPca-oppose, i.e., as the number of cache partitions used by the higher-priority tasks increases.

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 with static core-level cache allocation), nFPca (cache-aware non-preemptive global scheduling with dynamic task-
level cache allocation), and gFPca (cache-aware preemptive global 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 in Fig. 8.

We then constructed a reference taskset \( \tau_{ref} \) 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 \( 1 < i \leq n \). (We observed similar results when varying the number of tasks.)

![Fig. 8: Measured WCET vs. Number of cache partitions.](image)

**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 \( A_i = 4 \) (\( i > 1 \)). We set the WCET of each task to be an upper bound of the WCET measured under the assigned number of partitions\(^2\), i.e., \( e_1 = 500 \) 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 [16, 25]) and shared-cache isolation is achieved among tasks on different cores via static cache partitioning. However, this static approach cannot schedule the example workload. 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 \((A_1 = 16 \text{ and } A_i = 4, \text{ with } i > 1)\). 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>gFPca</th>
<th>gFP</th>
<th>nFPca</th>
<th>pFP</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0.8</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>0.7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0.6</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0.5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0.4</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0.3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0.2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</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 its implementation to improve their efficiency and performance.

**ACKNOWLEDGEMENT**

This research was supported in part by the ONR N00014-13-1-0802 and N00014-16-1-2195, NSF CNS-1117185, ECCS 1135630 and CNS 1329984.
REFERENCES


APPENDIX A: gFPca Run-time Overhead

We used the feather-trace tool to measure the overheads, as in earlier LITMUSRT-based studies (e.g., [9, 10]). 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 nanoseconds) to trace the Inter-Processor Interrupt (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 [9] 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><strong>gEDF</strong></td>
<td><strong>gFPca</strong></td>
</tr>
<tr>
<td>Release</td>
<td>5.72</td>
</tr>
<tr>
<td>Sched</td>
<td>8.64</td>
</tr>
<tr>
<td>CSS</td>
<td>4.23</td>
</tr>
<tr>
<td>IPI</td>
<td>4.06</td>
</tr>
</tbody>
</table>

TABLE 2: Average overhead (µs) under different schedulers.

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 LITMUSRT 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 [27].

APPENDIX B: 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 LITMUSRT 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 [27].
the same cache partition, we can tighten $\Delta_t$ by considering the cache partitions used by other tasks:

**Lemma 12.** The cache overhead a task $\tau_i$ experiences when it resumes from one preemption is upper bounded by $\Delta_t = \text{PRT} \times |\text{UCP}_i \cap \bigcup_{j \notin P(C)} \text{ECP}_j| \leq \text{PRT} \times \min\{|\text{UCP}_i|, \sum_{j \notin P(C)} |\text{ECP}_j|\}$. 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 13.** 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.

Note that the second condition in the above lemma holds because when a higher-priority task $\tau_i$ is released, it may need to preempt a currently running medium-priority task (to acquire the cache resources it needs), and this medium-priority task then releases enough cache resources not only for $\tau_i$ to execute but also for the low-priority task $\tau_i$ to resume.

The next lemma follows directly from Lemma 13.

**Lemma 14.** The maximum number of task-resumption events of $\tau_i$ during each period is at most $N_{\text{RS}} = \sum_{j \in \text{ECP}_i} |\text{ECP}_j| + 2$.

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

**Overhead-aware analysis:** Since the total overhead of $\tau_i$ is at most $N_{\text{RS}} \times \Delta_t$, the WCET of $\tau_i$ in the presence of cache overhead is at most $e_{\text{f}i} = e_i + N_{\text{RS}} \times \Delta_t$. As a result, the overhead-aware analysis can be established by applying the overhead-free analysis on the inflated workload.

**APPENDIX C: EXTENSION TO OTHER OVERHEAD TYPES**

Real-time tasks typically experience six major sources of overhead [8]: 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 $\Delta_{\text{rel}}, \Delta_{\text{sched}}, \Delta_{\text{cxs}}, \Delta_{\text{iwp}}, \Delta_{\text{crpmd}},$ and $\Delta_{\text{tck}}$. 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.) 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 $\tau_i$ 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 $\hat{e}_i = e_i + \Delta_{\text{rel}} + \Delta_{\text{iwp}} + \Delta_{\text{sched}} + \Delta_{\text{cxs}}$ is a safe bound on the execution time $e_i$ of $\tau_i$ in the presence of the overhead when the task executes alone.

**Overhead accounting under gFP.** When a preemption event of $\tau_i$ occurs under gFP, $\tau_i$ incurs one scheduling overhead, one context switch overhead, and one CRPMD overhead, similar to the preemption-related overhead scenario under gEDF shown in [8]. Since gFP does not provide cache isolation, concurrently running tasks may still evict out the cache content of each other. Since it is difficult to predict or analyze which cache content of a task may be evicted by another currently running task, we assume all cache accesses incur cache misses to safely account for the shared-cache overhead under gFP. Let $\alpha_0$ be the fraction of the WCET of a task $\tau_i$ that is spent on cache hit without the shared-cache interference, and $\text{hit\_latency}$ and $\text{miss\_latency}$ be the cache hit and miss latency of the shared cache, then the shared-cache overhead of $\tau_i$ under gFP is $\delta_k = (\alpha_0 e_i) / \text{hit\_latency} \times (\text{miss\_latency} - \text{hit\_latency})$. Therefore, the inflated execution time of $\tau_i$ that accounts for five types of overhead is bounded by $e_{\text{f}i} = e_i + \Delta_{\text{rel}} + \Delta_{\text{iwp}} + \Delta_{\text{sched}} + \Delta_{\text{cxs}} + \Delta_{\text{crpmd}} + \delta_k$.

**Overhead accounting under nFPca.** Because no preemption occurs under nFPca, the WCET of each task $\tau_i$ that accounts for all five types of overhead under nFPca is bounded by $e_{\text{f}i} = e_i + \Delta_{\text{rel}} + \Delta_{\text{iwp}} + \Delta_{\text{sched}} + \Delta_{\text{cxs}}$.

**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.