Co-Optimizing Cache Partitioning and Multi-Core Task Scheduling: Exploit Cache Sensitivity or Not?

Binqi Sun, Debayan Roy, Tomasz Kloda, Andrea Bastoni, Rodolfo Pellizzoni, Marco Caccamo

Co-Optimizing Cache Partitioning and Multi-Core Task Scheduling: Exploit Cache Sensitivity or Not?

Binqi Sun[1], Debayan Roy[1], Tomasz Kloda[2], Andrea Bastoni[1], Rodolfo Pellizzoni[3], Marco Caccamo[1]
[1]Technical University of Munich, Germany
[2]LAAS-CNRS, Université de Toulouse, INSA, Toulouse, France
[3]University of Waterloo, Canada
Email: {binqi.sun, debayan.roy, andrea.bastoni, mcaccamo}@tum.de, tomasz.kloda@laas.fr, rpellizz@uwaterloo.edu

Abstract*—Cache partitioning techniques have been successfully adopted to *mitigate interference* among concurrently-executing real-time tasks on multi-core processors. Considering that the execution time of a cache-sensitive task strongly depends on the cache available for it to use, *co-optimizing* cache partitioning and task allocation improves the *schedulability* of the system. In this paper, we propose a *hybrid multi-layer* design space exploration technique to solve this *multi-resource* management problem. We explore the *interplay* between cache partitioning and schedulability by systematically *interleaving* three optimization layers, viz., (i) in the outer layer, we perform a *breadth-first search* combined with proactive *pruning* for cache partitioning; (ii) in the middle layer, we exploit a *first-fit* heuristic for allocating tasks to cores; and (iii) in the inner layer, we use the well-known recurrence relation for the schedulability analysis of *non-preemptive fixed-priority (NP-FP) tasks* in a uniprocessor setting. Although we focus on NP-FP scheduling, we evaluate the flexibility of our framework in supporting different scheduling policies (NP-EDF, P-EDF) by plugging-in appropriate analysis methods in the inner layer. Experiments show that, compared to the state-of-the-art techniques, the proposed framework can *improve real-time schedulability of NP-FP task sets by 15.2% on average* with a maximum improvement of 233.6% (when tasks are highly cache-sensitive) and a minimum of 1.6% (when cache-sensitivity is low). For such task sets, we found that *clustering* similar-period (or *mutually compatible*) tasks often leads to higher schedulability (on average 7.6%) than clustering by *cache sensitivity*. In our evaluation, the framework also achieves good results for preemptive and dynamic-priority scheduling policies.

I. Introduction

Nowadays, heterogeneous multiprocessor system-on-a-chip (MPSoC) platforms are routinely used for all those workloads that require performance, real-time capabilities, and limited size and power consumption. These workloads include, *e.g.*, applications found in autonomous driving, intelligent robotics, and unmanned aerial vehicles domains. Towards guaranteeing real-time performance, these platforms pose an unprecedented challenge to the management of the memory hierarchy. With a focus on the core complex of such MPSoCs, sharing caches among cores prevents analyzing tasks in isolation, thus complicating an accurate estimation of the tasks' worst-case execution times (WCETs). Unsurprisingly therefore, to mitigate this problem, both software-based [1], [2], [3], [4] and hardware-based [5], [6] cache partitioning techniques have been exploited. Although effective, cache partitioning limits the amount of cache available to (groups of) real-time tasks. Therefore, the impact of cache partitioning on the WCET can be non-negligible for a *cache-sensitive* workload. This effect

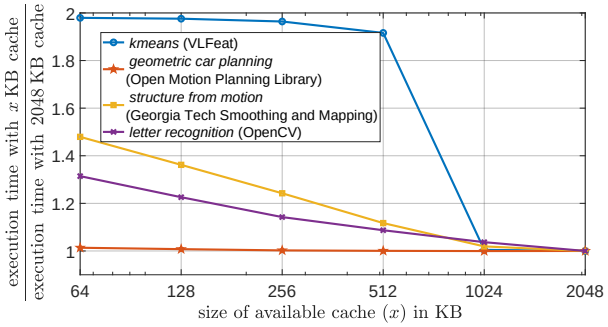


Fig. 1: Benchmark's execution slowdown with x KB cache compared to full (2048 KB) cache.

is illustrated in Figure 1, which reports the *slowdown* due to reduced cache availability of four benchmark applications (more details in Section VI). For example, *kmeans* is almost two times slower when it runs with a cache partition smaller than 256 KB instead of 1024 KB.

Problem setting: This paper studies the *integrated problem* of (1) assigning real-time tasks to cores and (2) reserving cache for tasks running on each core. The goal is to achieve a solution where the tasks are *schedulable*, *i.e.*, each task meets its real-time requirements, *e.g.*, deadline. The main focus of the proposed optimization strategies is on *non-preemptive fixed-priority* (NP-FP) scheduling. We further assume that tasks are *statically assigned* to cores. Partitioned schedulers have simpler implementations and generally lower overheads [7], and non-preemption naturally separates computation from data management phases (*e.g.*, [8]). Also, we assess the flexibility of our framework to support other scheduling policies such as preemptive and non-preemptive *earliest deadline first* (EDF).

Proposed framework: Given the *interdependencies* of the three sub-problems, viz., task allocation, cache partitioning, and schedulability analysis, this paper studies an *integrated* solution to improve the likelihood of establishing system schedulability. In particular, we propose a *nested multi-layer, hybrid* optimization framework to *explore the interplay* between the sub-problems. In this framework, (i) the outer layer partitions the shared cache, (ii) the middle layer allocates tasks, and (iii) the inner layer performs the schedulability analysis.

We perform a *polynomial-time breadth-first search* in the outer layer using a heuristic to *proactively prune* the search tree to prevent its exponential growth (Section IV-B). The outer layer chooses a cache partition size for a core and invokes the

middle layer to allocate tasks to the core from the remaining ones. We develop two strategies for task allocation. While one tries to cluster tasks that are *compatible* for co-scheduling, the other one co-allocates tasks with similar *cache sensitivity potentials* (Section IV-A). The schedulability of the task set allocated by the middle layer is checked by the inner layer using the exact method reported in [9] for NP-FP scheduling.

Although the selection of tasks is optimized for NP-FP scheduling, the multi-layer framework can be easily adapted to other scheduling policy by plugging-in an appropriate schedulability test in the inner layer. To demonstrate this, we show experimental results under *preemptive EDF* (P-EDF) and *non-preemptive EDF* (NP-EDF) scheduling by using the tests adopted by [5] and [10], [11], without any modifications to the outer and middle layers (Section VI-C).

Contributions: This paper has the following contributions:

- A generic multi-layer, hybrid optimization framework is proposed to solve the joint problem of cache partitioning and partitioned scheduling of real-time tasks. In particular, we systematically extend the first-fit heuristic for task allocation with an outer layer employing an intelligent breadth-first search for cache partitioning.
- To the best of our knowledge, this paper shows for the first time that the characteristics of task sets (including task periods and cache sensitivities) guide the choice of heuristics to solve the aforementioned problem.
- A metric is introduced to evaluate the cache sensitivity potential of a task that assists in task allocation. This metric captures the maximum-possible reduction in the utilization of a task if more cache can be offered.
- The framework is extensively evaluated and compared with multiple state-of-the-art techniques [5], [10], [11], using both benchmark-derived and synthetic cache slowdown profiles. Results for NP-FP scheduling indicate that the performance of the framework depends on the cache-sensitivity of workloads with a schedulability improvement of up to 14.5% for tasks with low cache-sensitivity and of up to 233.6% for highly cache-sensitive tasks, with an average improvement of 15.2%. NP-FP experiments also show that focusing on compatibility leads to better results (by 7.6% on average) than cache sensitivity. For P-EDF scheduling, the framework improves the schedulability by 8.7% on average compared to the approach in [5], while for NP-EDF scheduling, the average improvement is 19.2% compared to the techniques in [10], [11].

Paper organization: We present relevant previous works in Section II. We define the problem in Section III. We describe our proposed framework and heuristics in Section IV. In Section V, using illustrative examples, we show that none of the proposed heuristics dominates the other. Experimental results and interesting trends are discussed in Section VI. Section VII provides concluding remarks.

II. Related Works

Preemptive task-cache co-allocation: Full exploitation of multiprocessor platforms can be achieved only if the allocation of tasks and memory (*e.g.*, cache) resources is performed jointly. *Tunable* WCETs [12] can be elastically adjusted to take into account shared resource allocation and arbitration methods. For this model, mixed-integer linear programming (MILP) has been used to partition tasks, cache, and bandwidth, minimizing the overall system utilization. In [13], tasks, cache, and bandwidth co-allocation problem is solved using a MILP formulation and a knapsack-algorithm-based heuristic. [13] considers P-EDF scheduling while allocating dedicated cache partitions to tasks. Likewise, [14] considers recurrent tasks scheduled under P-EDF and proposes a different heuristic for task and cache allocation on a multiprocessor. Under partitioned P-EDF, the dependence of execution time on the number of available cache partitions has been studied in [5], [15], [16]. [5] uses k-means clustering and a first-fit heuristic to partition shared caches and allocate tasks on a multiprocessor based on cache sensitivities of tasks. [5] also compares the obtainable schedulability with the strategy adopted by [15] (and later also used in [16], [17]). The above works mainly consider setups for which exact utilization bound schedulability tests are available, which is not true for our setup. Nevertheless, in Section VI, we compare our framework with [5] with appropriate adaptations for non-preemptive and preemptive scheduling.

Non-preemptive task-cache co-allocation: For non-preemptive scheduling, the problem becomes more complex as there are no efficient utilization bounds to check the schedulability in polynomial time. The multi-resource allocation problem for time-triggered non-preemptive scheduling naturally fits into integer linear programming (ILP) formulation [18]. *Interference Aware Allocation Algorithm* (IA^3) [10] and *Period Driven Task and Cache Partitioning Algorithm* (PDPA) [11] are proposed for NP-EDF. These works are the closest to ours. While, qualitatively, we search the design space more thoroughly than them using a carefully designed pruning criterion and a cache sensitivity potential metric. In Section VI, we quantitatively compare our proposed approach to them (for both NP-FP and NP-EDF). Also, we empirically show that the characteristics of task sets guide the selection of the heuristics to solve the problem.

Cache-related preemption delays (CRPD): In multitasking systems, partitioning shared caches and allowing each task to run using specific partitions can reduce CRPD, but it may increase cache misses. [19] shows how to systematically compute CRPD. Optimally exploring the trade-off between CRPD and cache misses, even for a single-core, is NP-hard [20]. Several heuristics [21], [20] have been proposed in this context to optimize task set utilization. Branch and bound can speed up the search for optimal cache partitioning [22]. In the same vein, [23] considers sharing cache partitions among tasks running preemptively on the same core while allowing cache isolation between cores on a multiprocessor. The task

TABLE I: List of symbols.

Symbols	Description
$\mathcal{T}$	set of n tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_{n_\tau}\}$;
$\mathcal{C}$	set of processors $\mathcal{C} = \{C_1, C_2, \ldots, C_{n_c}\}$;
n_p	total number of cache partitions;
p_i	period of task τ_i;
$\epsilon_{i,\mu}$	execution time of τ_i with μ cache partitions;
T_j	set of tasks allocated to core C_j;
μ_j	number of cache partitions allocated to core C_j;
γ_i	cache sensitivity potential of task τ_i;
$\widehat{u}_i$	base utilization of task τ_i, $\widehat{u}_i = \epsilon_{i,n_p}/p_i$;
ω_{init}	an empty solution;
$\omega, \omega', \omega^+$	a (new) partial solution;
Ω^*	set of new partial solutions;
Ω_x	set of partial solutions at search depth x;
$\overline{\mathcal{T}^*}$	set of remaining tasks to be allocated;
$\overline{\mathcal{T}^*}$	sorted list of remaining tasks to be allocated;
U_R	scheduling demand of remaining tasks to be allocated;

and cache allocations are performed using best-fit decreasing bin-packing. The approach is also extended for multi-core virtualization [24]. Contrary to these works, we consider that only NP-FP tasks running on a core can share cache partitions and, hence, they do not experience CRPD.

Other approaches: Instead of partitioning shared caches to cores, [25] formulates an ILP to upper bound the inter-core cache interference and proposes a task partitioning algorithm based on the interference upper bound. Dynamic resource allocation has also been explored, *e.g.*, [26] adapts the resource allocation based on program phases, while [27], [28] dynamically allocate resource at mode transitions.

III. Problem Description

In this section, we describe the design problem under study. In particular, we want to (i) determine an allocation of software tasks to cores and (ii) identify the maximum portion of cache each task can use so that (iii) the tasks meet their respective deadlines. Table I summarizes the most important symbols used in the following sections.

A. Task allocation

We consider a set of n software tasks, denoted by $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_{n_\tau}\}$. These tasks need to run on n_c processing cores. We denote the set of processors by $\mathcal{C} = \{C_1, C_2, \cdots, C_{n_c}\}$. For this setting, we need to determine *how the tasks should be allocated to the cores*, which is the first part of the problem at hand. Hence, we determine $T_1, T_2, \cdots, T_{n_c}$, where T_j is a set of tasks that will run on the core C_j. We study *partitioned* multi-core scheduling, *i.e.*, a task $\tau_i \in T_j$ allocated to a core C_j will always be executed by C_j. It means that the task allocation is static. In mathematical terms, $\{T_1, T_2, \cdots, T_{n_c}\}$ is a partition of set τ.

B. Cache partitioning

We assume that tasks execute on cores that share a cache of size M (this model is common in current MPSoCs, *e.g.*, [29], [30]). To prevent memory interference, we consider dividing the shared cache into n_p partitions of equal size. The tasks assigned to a core can only use a certain number of partitions. Our approach applies to *any* cache-partitioning technique, either with hardware-support (eg, [31], [32]), or purely implemented in software (*e.g.*, [3], [4]).

We reserve only a certain number of cache partitions for each core and let the tasks running on a core use only the assigned partitions. Now, the amount of cache that the tasks running on a core optimally need depends on how their execution times vary with available cache and their real-time requirements. Hence, we need to determine *the appropriate number of cache partitions μ_j that should be made available to the tasks (in T_j) mapped on a core C_j*, which is the second part of the problem under consideration. Here, we cannot reserve the same partition for tasks mapped on different cores. Hence, we need to respect a constraint given by $\sum_{j=1}^{n_c} \mu_j \leq n_p$, *i.e.*, the total number of partitions in use cannot be more than n_p.

C. Task specification

We consider that each non-preemptive task $\tau_i \in \mathcal{T}$ is dispatched sporadically respecting a minimum time p_i between two consecutive dispatches. In this paper, we also refer to p_i as the period of the task τ_i. We study the case where the deadline of a task τ_i is exactly equal to its period p_i. The algorithms presented in this paper are valid or can be trivially extended for other deadline constraints as well. The execution time e_i of a task τ_i depends on the number of cache partitions it can use. Hence, we can write $e_i = \mathcal{E}_i(\mu)$, where $\mathcal{E}_i(\cdot)$ is a discrete function of the available number of cache partitions $\mu \in \{1, 2, \cdots, n_p\}$. We can write the range of $\mathcal{E}_i(\cdot)$ as an ordered set $(\epsilon_{i,1}, \epsilon_{i,2}, \cdots, \epsilon_{i,n_p})$, where $\epsilon_{i,\mu}$ is the execution time when τ_i uses μ cache partitions. Note that here we assume that a task will get to use at least one cache partition. In summary, we specify a task τ_i using its period (or minimum inter-arrival time) p_i and the function $\mathcal{E}_i(\cdot)$ capturing the variation of its execution time with the available number of cache partitions. We assume that p_i and $\mathcal{E}_i(\cdot)$ can be computed a priori for τ_i, and $\mathcal{E}_i(\cdot)$ includes scheduling overheads and all other sources of interference (*e.g., inter-core interference*). We note that different solutions can be used alongside to mitigate the timing interference in modern multi-core platforms (*e.g.,* bank partitioning [33], [34], [35], software bandwidth regulators [36], [37] or segmented execution models [38], [39], [40], [41], [42]).

D. Schedulability analysis

We need to *verify the schedulability of a set of tasks T_j* that is allocated to a core C_j and uses μ_j cache partitions to run, which is the third part of the problem under study. For a task $\tau_i \in T_j$, we can write $e_i = \epsilon_{i,\mu_j}$. We consider that a task cannot be preempted during execution. Besides, each task has a fixed priority according to the rate monotonic scheduling policy. A task τ_i has a higher priority than a task $\tau_{i'}$ (where $\tau_i, \tau_{i'} \in T_j$) if $p_i < p_{i'}$. When $p_i = p_{i'}$, we assume that τ_i has a higher priority than $\tau_{i'}$ if $e_i > e_{i'}$. In all other cases, τ_i has a lower priority than $\tau_{i'}$. Here, no two tasks mapped on the same core have the same priority.

We compute the worst-case response time of a task under the NP-FP scheduling policy using the technique outlined

in [9]. First, we determine the busy period t_i of a task $\tau_i \in T_j$ using the following recurrence relation:

$$t_i^{k+1} = B_{i,j} + \sum_{\tau_{i'} \in HEP_{i,j}} \left\lceil \frac{t_i^k}{p_{i'}} \right\rceil e_{i'} \quad (1)$$

where, $B_{i,j} = \max_{\tau_{i'} \in LP_{i,j}} e_{i'}$. Here, (i) $HEP_{i,j} \subseteq T_j$, $LP_{i,j} \subset T_j$ where $HEP_{i,j}$ and $LP_{i,j}$, respectively, comprise the tasks that have higher or equal and lower priorities than τ_i; and (ii) $B_{i,j}$ is the maximum time for which the task τ_i can be blocked by a lower priority task. To solve Equation 1, we start with $t_i^0 = e_i$ and continue until we get $t_i^{k+1} = t_i^k$. Here, the recurrence relation is guaranteed to converge if $\sum_{\tau_{i'} \in HEP_{i,j}} \frac{e_{i'}}{p_{i'}} < 1$. Further, we calculate the number of instances Q_i of τ_i that execute in its busy period t_i as follows:

$$Q_i = \left\lceil \frac{t_i}{p_i} \right\rceil. \quad (2)$$

We compute the response time for each of the Q_i instances. We can calculate the longest time $w_i(q)$ between the start of the busy period and the start of the execution of the q-th instance ($1 \leq q \leq Q_i$) of τ_i using the following recurrence relation:

$$w_i^{k+1}(q) = B_{i,j} + (q-1) \cdot e_i + \sum_{\tau_{i'} \in HP_{i,j}} \left\lceil \frac{w_i^k(q) + \delta}{p_{i'}} \right\rceil e_{i'}. \quad (3)$$

Here, (i) $HP_{i,j} \subset T_j$ comprises the tasks that have higher priorities than τ_i; and (ii) $\delta > 0$ is a very small number. To solve the recurrence relation in Equation 3, we start with $w_i^0(q) = B_{i,j} + (q-1) \cdot e_i$ and stop when $w_i^{k+1}(q) = w_i^k(q)$. The response time of the q-th instance of τ_i is given as follows:

$$R_i(q) = w_i(q) - (q-1) \cdot p_i + e_i. \quad (4)$$

The worst-case response time R_i^{wc} is computed as follows:

$$R_i^{wc} = \max_{1 \leq q \leq Q_i} R_i(q). \quad (5)$$

An implicit-deadline task τ_i meets its deadline if and only if

$$R_i^{wc} \leq p_i. \quad (6)$$

A task set is schedulable if and only if Equation 6 holds for each task in the task set.

IV. Multi-Layer Hybrid Optimization

As described in Section III, the problem under study comprises three interdependent parts. In this paper, we propose an integrated solution with three nested layers. The outer layer partitions the shared cache, the middle layer allocates tasks, and the inner layer performs schedulability analysis. Algorithm 1 outlines the middle and the inner layers, while Algorithm 2 captures the outer layer.

A. Algorithm 1: Middle and inner optimization layers

We provide a set of remaining tasks $\mathcal{T}^*$ and their timing attributes as input to Algorithm 1. The period p_i and the least-possible execution time ϵ_{i,n_p} (*i.e.*, when the whole cache is available) are known a priori for a task $\tau_i \in \mathcal{T}^*$. Further, in the inner two layers, we deal with a fixed number of cache partitions μ as chosen by the outer layer (more details on the outer layer in Sec. IV-B). Corresponding to μ, the execution time e_i of a task $\tau_i \in \mathcal{T}^*$ also gets a fixed value as $e_i = \epsilon_{i,\mu}$.

Algorithm 1: Middle and inner optimization layers

Input: $\{(p_i, e_i, \epsilon_{i,n_p}) | \tau_i \in \mathcal{T}^*\}$;
Output: T;
1 $T \leftarrow \varnothing$;
2 $\overline{\mathcal{T}^*} \leftarrow$ **Sort**$(\mathcal{T}^*)$;
/* different sorting criteria can be applied */
3 **for** $\tau_i \in \overline{\mathcal{T}^*}$ **do**
4 | **if isSchedulable**(T, τ_i) **then** // inner layer
5 | | T.**append**(τ_i);
6 **return** T;

The inner two layers select a set of tasks $T \subseteq \mathcal{T}^*$ that can be allocated to a core with a given amount of cache μ without violating schedulability constraints. Also, the goal is to increase the likelihood of scheduling the tasks in $\mathcal{T}^* \setminus T$ on the remaining cores with the rest of the cache. There are two major challenges here. First, the number of possible selections is exponential with respect to the number of tasks in $\mathcal{T}^*$ and, hence, it is computationally expensive to go through each of them. Second, it is non-trivial to identify a metric to optimize.

To elaborate on the second challenge, let us first define the base utilization $\widehat{u}_i = \frac{\epsilon_{i,n_p}}{p_i}$ of a task as the ratio of its execution time and period when the whole cache is available for it to use. Now, the scheduling demand U_R of the tasks remaining $\mathcal{T}^* \setminus T$ after Algorithm 1 has operated on $\mathcal{T}^*$ can be defined as the sum of the base utilizations of these tasks, *i.e.*,

$$U_R = \sum_{\tau_i \in \mathcal{T}^* \setminus T} \widehat{u}_i. \quad (7)$$

Intuitively, we would like to minimize U_R. However, *unlike in P-EDF scheduling*, utilization cannot be the only deciding factor. For example, let us consider that we can end up in two scenarios: (i) Two remaining tasks have the same period of 80 time units and base utilizations of 0.15 and 0.2, respectively. (ii) Two remaining tasks have periods of 10 and 80 time units, respectively, and base utilizations of 0.1 and 0.2, respectively. In scenario (ii), $U_R = 0.3$, while in scenario (i), $U_R = 0.35$. Hence, intuitively, we would like to be in scenario (ii). However, in this scenario, the two tasks are not schedulable on one core because the lower priority task might block the higher priority task for a time (at least 16 time units) greater than its deadline (10 time units). In the rest of the discussion, we term two tasks to be *mutually incompatible* if they are not schedulable together on a core despite their utilizations adding up to less than or equal to 1. When the number of remaining cores is more than one and there are several tasks yet to be allocated, it is not trivial to analyze how many mutually compatible schedulable task sets are formed by the remaining tasks.

Having established that mutual compatibility between remaining tasks also plays an important role in deciding their schedulability, we use the same criterion to select tasks as in the first-fit heuristic that works well for non-preemptive tasks [43]. That is, we sort the tasks in $\mathcal{T}^*$ in line 2 of the algorithm according to a non-decreasing order of their

periods. When our multi-layer optimization uses the above sorting criterion in the middle layer, we term it as *COMP*. We point out that unlike in a first-fit heuristic where only one subset of $\mathcal{T}^*$ is allocated to a core before moving to the next core, our optimization strategy efficiently explores multiple subsets where each corresponds to a different cache allocation.

Now, in lines 3 - 5, we iterate through the tasks in the sorted list $\overline{\mathcal{T}^*}$. In each iteration, we take a task and check if we can add it to the list of selected tasks T without jeopardizing the schedulability. Here, tasks with shorter periods will be allocated first. Hence, in later iterations of Algorithm 2 (the outer layer), when we will deal with tasks with longer periods and (likely longer) execution times, there is a high probability that they will *not* be mutually incompatible. Note that in the inner layer (*i.e.*, line 4), we check the schedulability of the tasks in T together with the new task τ_i using the exact analysis from [9] as outlined in Section III-D.

Should we exploit cache sensitivity while allocating tasks? Note that the execution time e_i of a task $\tau_i \in \mathcal{T}^*$ is fixed in the inner layers. Accordingly, the utilization $u_i = \frac{e_i}{p_i}$ of the task materializes only once it is selected by Algorithm 1 to be allocated to a core. We define cache sensitivity potential γ_i as the difference between the utilization of a task τ_i if it is selected by Algorithm 1 and its base utilization $\widehat{u}_i$, *i.e.*,

$$\gamma_i = u_i - \widehat{u}_i. \tag{8}$$

The lower the value of γ_i is, the lower is the cache sensitivity potential of τ_i. Intuitively, the "potential" expresses the possibility of considerably reducing e_i by providing more cache partitions. Hence, postponing the allocation of a task with a lower potential might not enhance the schedulability because we cannot reduce its utilization significantly.

From another perspective, let us consider two tasks τ_i and τ_2 with $\widehat{u}_1 = 0.15$, $\widehat{u}_2 = 0.2$, $\gamma_1 = 0.06$, and $\gamma_2 = 0.01$. Here, each task will use $21\,\%$ of the processor time and let us assume that we can allocate only one of them. When we allocate τ_1, the remaining scheduling demand U_R will be higher compared to that obtained by allocating τ_2. Hence, to follow the intuition of minimizing U_R, we would like to add tasks with lower values of cache sensitivity potentials. Keeping the template of Algorithm 1, we can sort the tasks in non-decreasing order of their cache sensitivity potentials. In this case, we term our multi-layer optimization as *CASE*.

Here, there is a possibility that mutually incompatible tasks might be left for later iterations. Besides, consider that we have allocated two tasks τ_1 and τ_2 on a core. τ_1 has $p_1 = 10$, $e_1 = 5$, and $\gamma_1 = 0$ and τ_2 has $p_2 = 25$, $e_2 = 5$, and $\gamma_2 = 0$. The utilization of the core is $70\,\%$. On the same core, if we want to allocate another task τ_3 with $p_3 = 10$ and $e_3 = 2$, we cannot do it. Now, consider another situation where, instead of τ_2, we have τ_4 that has $p_4 = 10$, $e_4 = 3$, and $\gamma_4 = 0.1$. Now, the utilization of the core is $80\,\%$. However, here, we can still add τ_3 to the core without violating schedulability. In the first case, the worst-case response time R_1 of τ_1 is 10 which is equal to its deadline. Thus, the blocking time for τ_1, *i.e.*, e_2, artificially increases the utilization of the core from $70\,\%$ to $100\,\%$, which is 1.5 times the utilization of τ_2 (*i.e.*, $20\,\%$). In Figure 1, we note that the execution time increases by up to $100\,\%$ for the benchmarks we have studied. In the second case, by allocating τ_4, we are compromising $10\,\%$ ($\gamma_4 = 0.1$) of the processor utilization assuming that we can reduce the execution time of τ_4 if we allocate it to another core with more cache partitions. Note that τ_4 is compatible with τ_1 and τ_3 as they have the same period. Again, we see that there is a trade-off between considering compatibility and cache sensitivity during task allocation. We will experimentally evaluate the relative dominance of these two factors in Section VI.

B. Algorithm 2: Outer optimization layer

In this layer, we focus mainly on cache partitioning for which we propose an algorithm that is loosely based on breadth-first search. Each node of the search tree represents a partial solution ω. A node ω at a depth x of the search tree comprises the following attributes: (i) *TaskAlloc* gives the sets of tasks $\{T_1, T_2, \cdots, T_x\}$ allocated to x cores. (ii) *CachePart* gives the number of cache partitions reserved for tasks allocated to each of the x cores, *i.e.*, $\{\mu_1, \mu_2, \cdots, \mu_x\}$. (iii) *TasksLeft* is the set of tasks that are yet to be allocated. (iv) *CacheLeft* represents the remaining number of cache partitions. (v) *RemSchedDemand* is the remaining scheduling demand that can be calculated based on *TasksLeft* using Eq. (7). Clearly, the root node $\omega_{init} \in \Omega_0$ (in line 1) has empty sets in *TaskAlloc* and *CachePart* respectively while *TasksLeft* comprises the complete task set $\mathcal{T}$, *CacheLeft* is equal to n_p, and *RemSchedDemand* can be computed as $\sum_{\tau_i \in \mathcal{T}} \widehat{u}_i$. At any time, Ω_x – x is the search depth – will store either the leaf nodes that represent a full solution or the parent nodes for which we will explore the child nodes.

Considering that we have n_c cores, the maximum depth of the search tree is n_c. In the x-th iteration of the for loop in lines 2 - 14, we explore the nodes at depth x. Ω^* (line 3) will store (i) the leaf nodes only if they represent a full solution and (ii) non-leaf nodes at depth x. Using the for loop in lines 4 - 13, we iterate through each node in Ω_{x-1}. If a node in Ω_{x-1} already represents a full solution, then it cannot have any valid child nodes and, hence, it is a leaf node. We can add such nodes directly to Ω^* (lines 12 - 13). Otherwise, we explore the child nodes of a node in Ω_{x-1}. Note that Algorithm 2 does not terminate when it finds a full solution. This is because our goal is also to find the solution that will reserve the minimum number of cache partitions to ensure the schedulability of the task set. Minimizing the number of cache partitions used for real-time tasks maximizes the cache available for soft real-time and best-effort tasks, thus also potentially improving the overall performance of the system.

For each parent node ω, we can have up to ω.*CacheLeft* number of child nodes. That is, in each iteration of the for loop in lines 6 - 11, we explore a child node (if valid). In the μ-th iteration, we consider that the tasks on the x-th core can use μ cache partitions. We invoke Algorithm 1 (the inner layers) in line 7 to obtain a set of tasks T_x to be allocated to the x-th core. If T_x is not empty, then we can create a new partial

Algorithm 2: Outer optimization layer

Input: $\mathcal{T}$, n_c, n_p;
Output: $\{T_1, T_2, \cdots, T_{n_c}\}$, $\{\mu_1, \mu_2, \cdots, \mu_{n_c}\}$;
1 $\Omega_0 \leftarrow \{\omega_{init}\}$;
2 **for** $x \leftarrow 1$ **to** n_c **do**
3 $\quad \Omega^* \leftarrow \varnothing$;
4 $\quad$ **for** $\omega \in \Omega_{x-1}$ **do**
5 $\quad\quad$ **if** $\omega.TasksLeft \neq \varnothing$ **then**
6 $\quad\quad\quad$ **for** $\mu \leftarrow 1$ **to** $\omega.CacheLeft$ **do**
$\quad\quad\quad\quad$ /* invoke Algorithm 1 to allocate tasks to the x-th core */
7 $\quad\quad\quad\quad$ $T_x \leftarrow$ **allocT**$(\omega.TasksLeft, \mu)$;
8 $\quad\quad\quad\quad$ **if** $T_x \neq \varnothing$ **then**
$\quad\quad\quad\quad\quad$ /* extend parent node ω to a new partial solution ω^+ */
9 $\quad\quad\quad\quad\quad$ $\omega^+ =$ **newPartSol**(ω, T_x, μ);
$\quad\quad\quad\quad\quad$ /* check if ω^+ can be extended to a full solution */
10 $\quad\quad\quad\quad\quad$ **if isProspectiveSolution**(ω^+) **then**
11 $\quad\quad\quad\quad\quad\quad$ Ω^***.append**(ω^+);
12 $\quad\quad$ **else**
13 $\quad\quad\quad$ Ω^***.append**(ω);
$\quad$ /* remove the dominated partial solutions */
14 $\quad$ $\Omega_x \leftarrow$ **removeDominatedPartialSolutions**(Ω^*)
/* return non-dominated full solution if any */
15 **if** $\Omega_{n_c} \neq \varnothing$ **then**
16 $\quad$ **return** $\{\Omega_{n_c}[1].TaskAlloc, \Omega_{n_c}[1].CachePart\}$;
17 **else**
18 $\quad$ **return** $\{\varnothing, \varnothing\}$;

solution ω^+ extending one of the parent nodes (lines 8 - 9). Note that for the parent node, we have the task allocation and cache partitioning until $x-1$ cores and we can now add T_x and μ to obtain ω^+. We further do some proactive pruning if possible (lines 10 - 11). That is, if ω^+ still have tasks waiting to be allocated while there are no cores or cache partitions left, then we only have a leaf node with an incomplete solution ω^+. There is no point in adding such a node to Ω^*.

After we explore the nodes at depth x, we *prune* the search tree further based on a heuristic (in line 14). We delete a node $\omega' \in \Omega^*$ if it is dominated by another node $\omega \in \Omega^*$. Here, ω dominates ω' if one of the following conditions is satisfied.

1) $\omega.CacheLeft > \omega'.CacheLeft$ and $\omega.RemSchedDemand \leq \omega'.RemSchedDemand$.
2) $\omega.CacheLeft = \omega'.CacheLeft$ and $\omega.RemSchedDemand < \omega'.RemSchedDemand$.

If $\omega.CacheLeft = \omega'.CacheLeft$ and $\omega.RemSchedDemand = \omega'.RemSchedDemand$, we keep just one of them for further exploration and remove the other(s). The intuition behind the heuristic is as follows: If ω' already uses an equal or more number of cache partitions than ω and still have more scheduling demand to meet, then there is a lower probability that a child of ω' will be a better solution than one of the children of ω. This pruning is necessary to keep the search tractable otherwise the tree might grow exponentially. With this pruning heuristic, before exploring the child nodes at depth $x > 1$, we can have only up to $n_p + 2 - x$ parent nodes where each uses a different number of cache partitions.

In the end, in lines 15 - 16, Algorithm 2 returns the non-dominated complete solution if it has found one (*i.e.*, if Ω_{n_c} is not empty) otherwise, in lines 17 - 18, it returns empty sets for task allocation and cache partitioning, respectively.

C. Complexity analysis

In Algorithm 2, we have three nested for-loops. The outer loop (lines 2 - 14) iterates for n_c times. The middle loop (lines 4 - 13) iterates for at most n_p times. This is because, as mentioned earlier, Ω_x can have a maximum of n_p+2-x nodes at the beginning of the x-th iteration of the outer loop. The inner loop (lines 6 - 11) also iterates up to n_p times. Hence, the number of times we invoke Algorithm 1 from Algorithm 2 is upper-bounded by $n_c \cdot n_p^2$. Now, in Algorithm 1, we have only one loop (lines 3 - 6) that iterates at most n_τ times. Thus, the number of times we invoke the schedulability analysis in the inner layer (line 4) is upper-bounded by $n_c \cdot n_p^2 \cdot n_\tau$.

Further, we evaluate the time complexity of the response time analysis for NP-FP tasks. The response time analysis of task τ_i must cover its entire busy period. The busy period t_i of the task τ_i must satisfy Equation (1) and we can write:

$$t_i = B_{i,j} + \sum_{\tau_{i'} \in HEP_{i,j}} \left\lceil \frac{t_i}{p_{i'}} \right\rceil e_{i'} \tag{9}$$

Using the following relation:

$$\frac{t_i + p'_i}{p'_i} > \left\lceil \frac{t_i}{p'_i} \right\rceil$$

we can upper bound the left-hand side of Equation (9) by:

$$\begin{aligned} t_i &< B_{i,j} + \sum_{\tau_{i'} \in HEP_{i,j}} \frac{t_i + p_{i'}}{p_{i'}} e_{i'} \\ &= B_{i,j} + t_i \sum_{\tau_{i'} \in HEP_{i,j}} \frac{e_{i'}}{p_{i'}} + \sum_{\tau_{i'} \in HEP_{i,j}} e_{i'} \\ &\leq \sum_{\tau_{i'} \in T_j} e_{i'} + t_i U_j \end{aligned}$$

where $U_j = \sum_{\tau_{i'} \in T_j} e_{i'}/p_{i'}$ is the total utilization of task set T_j. By rearranging the terms, the busy period t_i of each task $\tau_i \in T_j$ can be upper-bounded as follows:

$$\forall \tau_i \in T_j : \; t_i \; < \; \frac{\sum_{\tau_{i'} \in T_j} e_{i'}}{1 - U_j} \; \leq \; \frac{n_\tau \max_{\tau'_i \in T_j} e_{i'}}{1 - U_j} \tag{10}$$

In each iteration of Equation (3), we need at most n_τ operations where the value of $w_i(q)$ increases by at least $\min_{i \in T_j} e_i$ time units (otherwise remains constant) within the busy period. The number of operations to compute task's response time is, hence, upper bounded by:

$$\frac{\max_{i \in T_j} e_i}{\min_{i \in T_j} e_i} \cdot \frac{n_\tau^2}{1 - U_j} \tag{11}$$

Taking into account the time-complexity of the response time analysis, the overall algorithm's (Algorithm 1 and 2) asymptotic complexity can be expressed as follows:

$$\mathcal{O}\left(\frac{n_c \cdot n_p^2 \cdot n_\tau^3}{1 - U_j} \cdot \frac{\max_{\tau_i \in T_j} e_i}{\min_{\tau_i \in T_j} e_i} \right). \tag{12}$$

TABLE II: *COMP* $\succ$ *CASE*

τ_i	τ_1	τ_2	τ_3	τ_4
p_i	100	100	150	150
$\epsilon_{i,1}$	36	75	77	85
$\epsilon_{i,2}$	35	55	48	82
$\epsilon_{i,3}$	34	45	35	81
$\epsilon_{i,4}$	34	27	25	79

TABLE III: *COMP* $\prec$ *CASE*

τ_i	τ_1	τ_2	τ_3	τ_4
p_i	200	200	250	250
$\epsilon_{i,1}$	35	177	324	65
$\epsilon_{i,2}$	33	172	178	63
$\epsilon_{i,3}$	31	168	119	62
$\epsilon_{i,4}$	26	165	80	60

Considering that we can put a limit on the utilization of a processing core, *e.g.*, $U_j = \sum_{\tau_i \in T_j} \frac{e_i}{p_i} < 0.99$, our optimization technique has a pseudo-polynomial time complexity.

V. Illustrative Examples: COMP vs CASE

Neither heuristics, *COMP* or *CASE*, completely dominate the other in terms of finding schedulable solutions. We illustrate this using two examples.

A. COMP *works,* CASE *fails*

Consider an example with 4 tasks, 4 cache partitions, and 2 cores. Table II gives the period of each task and shows how the execution time varies with the number of available cache partitions. To this task set, we first apply *COMP* to obtain the following schedulable solution:

$$\mu_1 = 2;\ \ T_1 = \{\tau_1, \tau_2\};\ \ \mu_2 = 2;\ \ T_2 = \{\tau_3, \tau_4\}.$$

It can be observed that, here, tasks with the same period are co-allocated to a core.

Now, let us try *CASE* on the same example. At a search depth of 1, we have two nodes in Ω_1 as follows: (i) ω_1: τ_1 is allocated to the first core and it uses 1 partition. (ii) ω_2: τ_1 and τ_3 are allocated to the first core and they share 2 partitions. For ω_1, we have only one task in a core with $(p_1, e_1) = (100, 36)$, which is schedulable. For ω_2, we have two tasks with $(p_1, e_1) = (100, 35)$ and $(p_3, e_3) = (150, 48)$ and, accordingly, $R_1^{wc} = R_2^{wc} = 83 < 100 < 150$, *i.e.*, the tasks are schedulable. Note that *CASE*, in this example, could not co-allocate τ_1 and τ_4 to a core despite having similar cache sensitivity potentials because they are not compatible. Further, instead of τ_2, τ_3 was selected after τ_1 in ω_2 because $\gamma_3 = 0.1533$ which is less than $\gamma_2 = 0.28$. Now, following ω_1, we cannot schedule the other 3 tasks on a single core sharing 3 remaining partitions because their utilization is greater than 1 (as shown before in *COMP*). Also, with respect to ω_2, τ_2 and τ_4 cannot be co-allocated to a core sharing 2 partitions because their combined utilization ($\frac{55}{100} + \frac{82}{150}$) is greater than 1. Hence, *CASE* cannot establish schedulability unlike *COMP*. This shows that there can be a task set for which *CASE* cannot obtain a schedulable solution while *COMP* can.

B. CASE *works,* COMP *fails*

Consider another example with task specification as given in Table III. Let us first study a case where we partition the cache equally, *i.e.*, we provision 2 cache partitions for the tasks running on each core. Corresponding to this, the execution time of each task gets fixed. Thus, we have to schedule 4 tasks on 2 cores with the following specification:

$$(p_1, e_1) = (200, 33);\quad (p_2, e_2) = (200, 172);$$
$$(p_3, e_3) = (250, 178);\quad (p_4, e_4) = (250, 63).$$

We can write the utilization of the tasks as follows:

$$u_1 = 0.165;\ \ u_2 = 0.86;\ \ u_3 = 0.712;\ \ u_4 = 0.252.$$

It is obvious that no task can be allocated together with τ_2 to a core because it will lead to a core utilization greater than 1. Alternatively, if we allocate the other 3 tasks (τ_1, τ_3, and τ_4) to a core, the total utilization will become $0.165{+}0.712{+}0.252 = 1.129 > 1$, which is again infeasible. Hence, this task set is not schedulable if we consider equal cache partitioning.

Now, when we apply *CASE* to this example, we get the following schedulable solution:

$$\mu_1 = 3;\ \ T_1 = \{\tau_1, \tau_3, \tau_4\};\ \ \mu_2 = 1;\ \ T_2 = \{\tau_2\}.$$

This example, therefore, illustrates that exploring the design space for cache partitioning together with task allocation and scheduling (using our proposed framework) can improve the likelihood of establishing the schedulability of a task set, which is the main motivation behind this work.

Further, we apply *COMP* to the same example. We have two nodes in Ω_1 at a search depth of 1. (i) ω_1: τ_1 and τ_4 are allocated to the first core and share 1 partition. Thus, we get $R_1^{wc} = R_4^{wc} = 100 < 200 < 250$. (ii) ω_2: τ_1 and τ_2 are allocated to the first core and share 3 partitions. Here, we get $R_1^{wc} = R_2^{wc} = 199 < 200$. In case of ω_1, the two remaining tasks are τ_2 and τ_3. They are not co-schedulable on a core even with 3 remaining partitions because their combined utilization is $\frac{168}{200} + \frac{119}{250} > 1$. In case of ω_2, the remaining tasks are τ_3 and τ_4. They cannot be mapped to one core and share 1 partition because their utilizations add up to $\frac{324}{250} + \frac{65}{250} > 1$. Hence, with *COMP*, we could not obtain a solution. Thus, this example shows that *CASE* can obtain a schedulable solution in certain cases where *COMP* cannot.

VI. Experimental Results

In this section, we will first explain different scenarios for the experiments and, then, we will present the experimental results and discuss our observations.

A. Design of experiments

Benchmark study: We first experimentally investigate the relation between a task's execution time and the size of available cache for real-world benchmarks. These benchmarks include computer vision applications from Georgia Tech Smoothing and Mapping (*e.g.*, structure from motion) [44], VLFeat (*e.g.*, k-means clustering) [45], and OpenCV (*e.g.*, letter recognition) [46] as well as motion planning algorithms from Open Motion Planning Library (*e.g.*, geometric car planning) [47].[1] We simulate the execution of each benchmark using *Cachegrind*[2] instrumentation tool and measure the number of instructions executed I, data cache hits $DH(\mu)$ and data cache misses $DM(\mu)$ for data cache sizes μ between 1

[1]The complete set of benchmark task has been uploaded as supplemental material and will be made available as technical report.

[2]https://valgrind.org/info/tools.html/#cachegrind

and 8192 KB. In Cachegrind, we have specified an 8-way set-associative last-level cache and a cache line size of 64 bytes. Note that Cachegrind only supports the *Least Recently Used* replacement policy and the number of sets is restricted to be a power of two.

We estimate the execution time $\mathcal{E}(\mu)$ of a given benchmark program for an available cache of size μ as follows:

$$\mathcal{E}(\mu) = I \cdot CPI + DM(\mu) \cdot MP + DH(\mu) \cdot HP, \quad (13)$$

where CPI is the number of clock cycles per instruction, MP and HP are the cache miss and hit penalties, respectively, in terms of the number of processor cycles (all parameters are platform-specific). We assume a superscalar processor that can execute $1/CPI$=2 instructions per cycle, and last-level cache hit and miss penalties of HP=20 and MP=200 cycles.

The proposed technique requires as input the execution time function for different cache sizes. We note that any method can be used to obtain it, including WCET analysis tool [48], [49], [50] or measurement-based approaches [51]. Our Cachegrind-based technique enables a fine-granular characterization of the dependence of execution time on the number of available cache partitions. The slowdown profiles are compatible with the slowdown obtained by both simulation methods and real observations in previous works [3], [5], [27], [52], [16].

We experimented with a wide variety of benchmarks (in terms of how much slowdown they can suffer) and show how varying slowdown affects the results. Although more than 50 benchmark programs have been analyzed, for our further experiments, we use the four most representative ones, *i.e.*, with a good spread of maximum slowdown values. The *slowdown* in the execution of these four benchmarks as a function of available cache size between 64 KB and 2048 KB is shown in Figure 1.

Synthetic slowdown profiles: Besides the slowdown profiles of benchmark programs, we use 8 synthetic profiles where the execution time decreases exponentially with the number of cache partitions. We can write these profiles as follows:

$$\frac{\epsilon_{i,\mu}}{\epsilon_{i,n_p}} = \frac{\exp(-\mu\alpha)}{\exp(-n_p\alpha)} \quad \text{where,}$$

$$\alpha \in \{0, 0.023, 0.036, 0.045, 0.052, 0.058, 0.067, 0.0743\}.$$

Recall that $\epsilon_{i,\mu}$ is the execution time of the task when it can use only μ cache partitions and ϵ_{i,n_p} is the execution time when it can use all cache partitions. Note that the values of α are selected to obtain maximum slowdowns of 1, 2, 3, 4, 5, 6, 8, and 10 when $n_p = 32$. For $n_p = 16$, the maximum slowdown values are 1, 1.4, 1.7, 2, 2.2, 2.4, 2.7, and 3. We note that such slowdowns can occur in practice for real workloads as reported in [53], [54]. We denote the profiles as $\mathcal{P}_1^{syn}, \mathcal{P}_2^{syn}, \cdots, \mathcal{P}_8^{syn}$ with respect to the values of α in ascending order.

Cache configurations: Inspired by MPSoC architectures with different cache-sizes, we consider two scenarios $\mathbb{AR}$-I and $\mathbb{AR}$-II, with four cores and a maximum number of partitions $n_p = 16$ and $n_p = 32$, respectively.

Test case generation: For generating a test case, we consider a system with 4 cores and 40 tasks. We use [55] to randomly synthesize base utilization of tasks for a target utilization U_{tar} where, $U_{tar} = \sum_{\tau_i \in \mathcal{T}} \widehat{u}_i$. For each $U_{tar} \in \{1.0, 1.1, \cdots, 3.9, 4\}$, we generate 100 task sets. To each task in a task set, we assign a slowdown profile (variation of execution time/utilization with available cache) either similar to an evaluated benchmark program—linearly interpolating for the non-available cache sizes—or generated synthetically.

Selections of slowdown profiles: We take three different sets of slowdown profiles. (i) $\mathbb{SD}$-B comprises slowdown profiles of four benchmark programs as depicted in Figure 1. (ii) $\mathbb{SD}$-S1 comprises synthetic slowdown profiles $\mathcal{P}_1^{syn}$, $\mathcal{P}_2^{syn}$, $\mathcal{P}_3^{syn}$, $\mathcal{P}_4^{syn}$, $\mathcal{P}_5^{syn}$, and $\mathcal{P}_6^{syn}$. (iii) $\mathbb{SD}$-S2 also includes synthetic profiles with high slowdown, specifically: $\mathcal{P}_1^{syn}$, $\mathcal{P}_2^{syn}$, $\mathcal{P}_4^{syn}$, $\mathcal{P}_6^{syn}$, $\mathcal{P}_7^{syn}$, and $\mathcal{P}_8^{syn}$.

Selections of task periods: We consider two different sets of values from which we choose task periods: (i) $\mathbb{WD}$: This set comprises a wider range of values. Each task $\tau_i \in \mathcal{T}$ is randomly assigned a period from $\{5, 10, 20, 40, 60, 80, 100\}$. Further, we do not limit the task utilization. Thus, in this case, two tasks with a wide gap in their periods, *e.g.*, 5 ms and 100 ms, can easily be incompatible for co-scheduling. (ii) $\mathbb{SH}$: This set comprises a shorter range of values, *i.e.*, we select task periods from $\{10, 15, 20, 25\}$. Moreover, we limit the base utilization of a task to 0.2 in this case. Considering that the gap between two task periods is not very wide and the utilization of a low-priority task may not be very high, when we select periods from $\mathbb{SH}$, the likelihood of two tasks cannot be scheduled together in a core is low.

Scenarios for experiments: In summary, we consider two cache configurations, $\mathbb{AR}$-I and $\mathbb{AR}$-II, with 16 and 32 partitions, respectively. Further, we have three selections of slowdown profiles, *i.e.*, $\mathbb{SD}$-B, $\mathbb{SD}$-S1, and $\mathbb{SD}$-S2. Also, we have two sets of task periods, *i.e.*, $\mathbb{WD}$ and $\mathbb{SH}$. In combination, we have 12 different scenarios as given in Tables IV-VI.

Baseline algorithms: To the best of our knowledge, no previous algorithm has been specifically proposed for co-optimizing cache partitioning and task allocation under NP-FP scheduling policy. In our experiments, we compare *CASE* and *COMP* against three state-of-the-art algorithms, *i.e.*, *CaM* [5], *IA*^{3} [10], and *PDPA* [11]. Since these algorithms were initially proposed for various scheduling policies (*CaM* for P-EDF, *IA*^{3} and *PDPA* for NP-EDF), we have adapted them preserving their allocation strategy, but using the NP-FP schedulability test. Conversely, in Section VI-C, *CASE* and *COMP* have been modified to suit P-EDF and NP-EDF. In the following, we first present the evaluation under NP-FP, which is the primary goal of the paper. Then, we demonstrate the flexibility of the optimization framework by comparing with the scheduling policies originally targeted by each baseline algorithms. Except for considering different schedulability test, the baseline algorithms were not otherwise modified.[3]

[3]Similarly to *e.g.*, [5], when considering preemptive scheduling, we assume that CRPD are already included in the WCET of tasks.

TABLE IV: Total number of schedulable task sets in each scenario using different algorithms under NP-FP. In **Bold** (*italic*), the results of the best-performing algorithm (best-performing baseline).

Scenario	*COMP*	*CASE*	*IA*^{3}	*PDPA*	*CaM*
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-B	**1954**	1889	*1887*	724	1768
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-S1	**1558**	1523	*1450*	863	1408
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-S2	**1302**	1280	*1153*	736	1142
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-B	**1981**	1632	*1950*	608	169
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-S1	**1564**	1335	*1475*	615	140
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-S2	**1293**	1101	*1185*	480	78
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-B	2356	**2434**	2064	1357	*2254*
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-S1	760	**832**	501	379	*584*
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-S2	515	**628**	154	124	*261*
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-B	**2348**	2014	*2050*	1105	287
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-S1	**801**	664	*521*	253	20
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-S2	**497**	407	*149*	64	1

B. Experimental results under NP-FP

The overall schedulability results under NP-FP scheduling policy are reported in Table IV. Besides, we illustrate the results of six representative scenarios in Figure 2.[4]

***COMP* vs baselines:** In all scenarios, *COMP* performs better than the three baseline algorithms, *IA*^{3}, *PDPA*, and *CaM*, in terms of schedulability (see Table IV and Figure 2). On average, *COMP* performs 13.5% better than the best-performing baseline. Although *COMP* does not cluster together tasks with similar cache sensitivities, it can improve schedulability by co-optimizing cache partitioning and task allocation. When the tasks have low cache sensitivities (a maximum slowdown of 3x in the scenarios involving $\mathbb{AR}$-I), *COMP* improves schedulability over the best-performing baseline by 6.1% on average (Table IV). We get a minimum gain of 1.6% when the maximum slowdown is 1.97x (*i.e.*, using $\mathbb{SD}$-B) and task periods are selected from the wider range of values (*i.e.*, $\mathbb{WD}$). Conversely, when the tasks have high cache sensitivities (up to 10x slowdown with $\mathbb{SD}$-S2 and $\mathbb{AR}$-II), *COMP* can schedule 97.3% (for $\mathbb{SH}$) and 233.6% (for $\mathbb{WD}$) more task sets. This clearly shows the opportunity to optimize schedulability with highly cache-sensitive workloads.

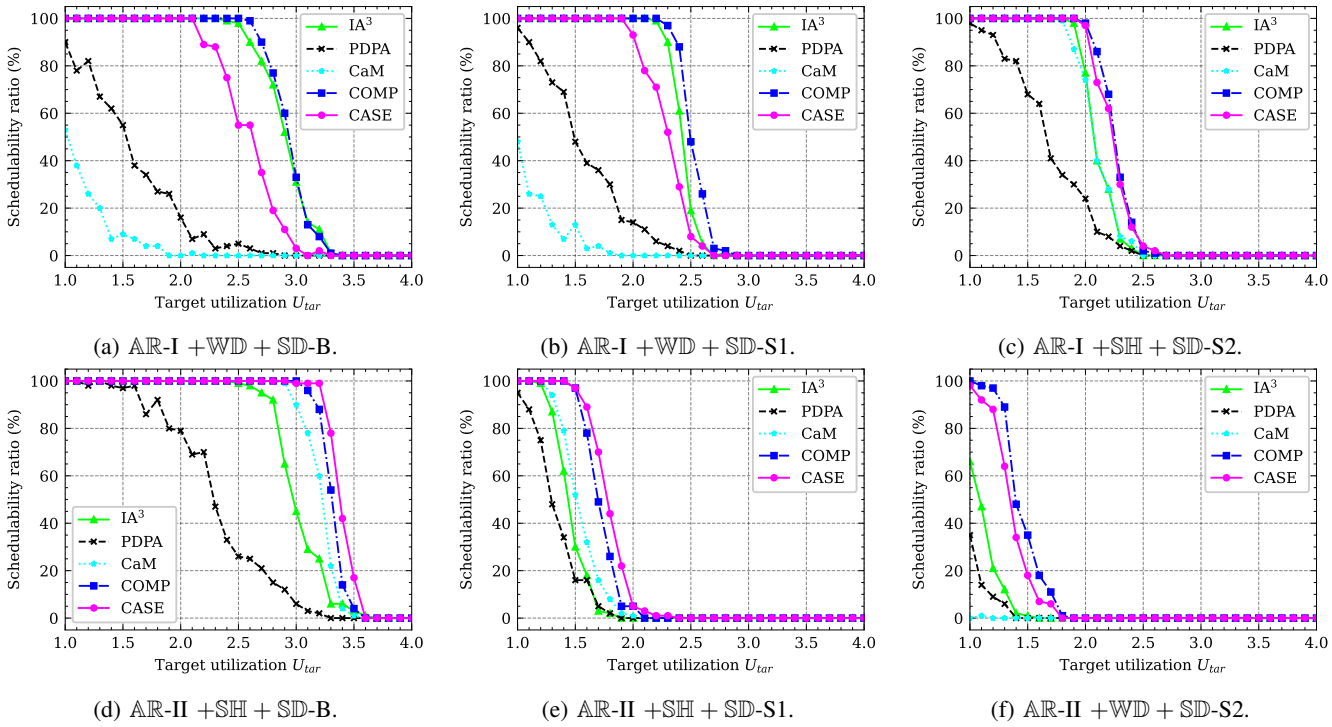
***CASE* vs baselines:** *CASE*, *IA*^{3}, and *CaM* consider cache sensitivity as main metric during allocation. For all scenarios where tasks have high cache sensitivity (maximum slowdown larger than 3x), *CASE* performs better than *CaM* and *IA*^{3} (see Table IV and Figures 2c-2f). However, *CASE* can perform worse for tasks with low cache sensitivities (only up to 3x slowdown) and periods in a wider range $\mathbb{WD}$ (*i.e.*, the ratio between two task periods can be as high as 20). In such cases, the gain for clustering tasks with similar cache sensitivities is limited and easily dominated by mutual incompatibility issues. *CASE* performs 16.3% worse than the best-performing baseline (*i.e.*, *IA*^{3}) in scenario $\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-B, but it can schedule 140.6% more task sets than the best-performing baseline (*i.e.*, *CaM*) in scenario $\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-S2. *CASE* is especially useful in the latter scenario, where tasks are highly cache-sensitive (up to 10x slowdown) and mostly compatible (periods are in a shorter range). On average, *CASE* performs 5.5% better than the best-performing baseline.

***COMP* vs *CASE*:** In our experiments, *CASE* has higher schedulability than *COMP* when both cache sensitivity of tasks is high and task periods are in a shorter range, *i.e.*, $\mathbb{AR}$-II + $\mathbb{SH}$ (see Table IV and Figures 2d-2e). Here, with $\mathbb{SD}$-S2 (up to 10x slowdown), *CASE* schedules 21.94% more task sets than *COMP* (Table IV). In scenarios with low cache sensitivity and shorter range of task periods (*e.g.*, $\mathbb{SH}$ + $\mathbb{SD}$-B), the average difference between *CASE* and *COMP* is smaller. In scenarios with wider period range ($\mathbb{WD}$), *COMP* performs at least 16% better than *CASE* (see Table IV and Figures 2a, 2b, 2f). The maximum improvement is 21.38% in scenario $\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-B (see Table IV and Figure 2a). Overall, in our experiments, *COMP* performs 7.56% better than *CASE*. An optimal use of our framework should therefore adopt *CASE* when tasks have high cache sensitivities and their periods are in a shorter range. Otherwise, it should use *COMP*. If we run *COMP* or *CASE* based on the characteristics of the task set—as identified from the results—our optimization framework can schedule 15.2% more task sets than the baselines. It is also trivial to run both *COMP* and *CASE* in parallel for a task set and select the highest schedulability.

Comparison among baselines: In 9 out of 12 scenarios, *IA*^{3} performs better than the other baselines. In the remaining three scenarios (high cache-sensitivity and short period range), *CaM* achieves the best performance (Table IV). Both *IA*^{3} and *CaM* use first-fit heuristics and explore cache sensitivity, but are designed in different ways: *CaM* clusters the whole task set into subsets with different slowdown profiles before applying the first-fit heuristic; *IA*^{3} allocates tasks with high cache sensitivity to the same core only when they are not schedulable by the standard first-fit heuristic. As a result, *CaM* favors task sets where cache sensitivity dominates task period, (*i.e.*, high cache-sensitivity and short period range). In our experiments, *PDPA* performs worse than other baselines in 9 out of 12 scenarios.

This is due to multiple reasons. First, *PDPA* attempts to initially assign one critical task to each core by selecting among the tasks with high utilization and low cache variability. After the critical tasks are assigned, other tasks can only be mapped to the core hosting a critical task with a higher or equal period. However, the algorithm does not guarantee that the task with the largest period is always selected as a critical task. In such cases, the tasks with periods higher than all critical tasks cannot be scheduled to any core. Our implementation of *PDPA* tries to avoid this situation by first selecting the task with the highest period as a critical task and then assigning it to the first core before selecting other critical tasks. Second, *PDPA* has a hard constraint (*i.e.*, line 9 of Algorithm 2 in [11]) to ensure that the periods of critical tasks are as far as possible from each other. [11] reports that $\delta = \frac{P_{max}-P_{min}}{M} \times \frac{\Delta=90}{100}$ produces the best performance in the experiments presented in that paper. The parameter $\Delta = 90$ does not work with our task sets since

[4]All schedulability plots have been uploaded as supplemental material and will be made available as a technical report.



(a) $\mathbb{AR}$-I $+\mathbb{WD}+\mathbb{SD}$-B. (b) $\mathbb{AR}$-I $+\mathbb{WD}+\mathbb{SD}$-S1. (c) $\mathbb{AR}$-I $+\mathbb{SH}+\mathbb{SD}$-S2.

(d) $\mathbb{AR}$-II $+\mathbb{SH}+\mathbb{SD}$-B. (e) $\mathbb{AR}$-II $+\mathbb{SH}+\mathbb{SD}$-S1. (f) $\mathbb{AR}$-II $+\mathbb{WD}+\mathbb{SD}$-S2.

Fig. 2: Schedulability ratio (*i.e.*, $\frac{\text{\#schedulable task sets}}{\text{\#total task sets}} \times 100$ %) of proposed schemes and baseline algorithms under NP-FP.

in some cases, no tasks can satisfy the constraint and thus the number of critical tasks is smaller than the number of cores. In our experiments, we therefore reduced $\Delta = 50$ so that one critical task is assigned to each core. Third, after generating an initial task allocation based on the critical tasks, *PDPA* only remaps selected tasks if a core is not schedulable even with the full cache. However, the algorithm does not check whether the total cache allocation is larger than the available cache size. Additionally, if every core is schedulable with the full cache, *PDPA* will not try to reduce the cache allocation to obtain a valid solution. However, as reported in [11], when task sets are restrictively generated to achieve minimum execution time for 1 to 4 cache partitions, *PDPA* might perform well.

Minimize cache reservation: Besides improving schedulability, our proposed schemes use less cache compared to baseline algorithms for almost all task sets. Let us denote $\mu_{\mathcal{T}}(\mathcal{A}) = \sum_{C_j \in \mathcal{C}} \mu_j$ the total number of cache partitions reserved for a task set $\mathcal{T}$ by an algorithm $\mathcal{A}$. The minimum between $\mu_{\mathcal{T}}(COMP)$ and $\mu_{\mathcal{T}}(CASE)$ is denoted by $\mu_{\mathcal{T}}(PROP)$. For the baseline algorithms, after the algorithm terminates and if the task set is schedulable, we go through the task allocation on each core and try to reduce the number of cache partitions reserved for the tasks without jeopardizing their schedulability. After this post-processing step, $\mu_{\mathcal{T}}(BASE)$ is the minimum cache partitions used among all baseline algorithms. If a task set $\mathcal{T}$ can be feasibly scheduled by a proposed strategy and a baseline algorithm respectively, we compute the number of cache partitions saved as $\mu_{save} = \mu_{\mathcal{T}}(BASE) - \mu_{\mathcal{T}}(PROP)$. On average, with $\mathbb{AR}$-I (16 available partitions), we can save 0.73 partitions, while with $\mathbb{AR}$-II (32 available partitions), we can save 3.68 partitions. Thus, we can save ca. 8.0% of the available partitions on average, which is significant. Non-real-time tasks can benefit from this saving without jeopardizing real-time performance. Figure 3 shows histograms for two scenarios with benchmarks profiles $\mathbb{SD}$-B, wider periods $\mathbb{WD}$, and 16/32 cache partitions. For certain task sets, we can save up to 5 (10) cache partitions out of 16 (32) available partitions, which is more than 31% of the cache size.

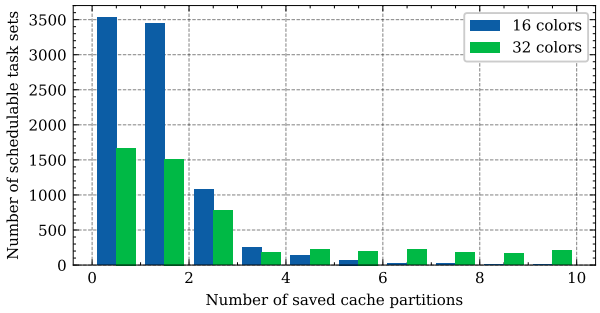


Fig. 3: Proposed schemes reserve less cache than baselines.

C. Experimental results under P-EDF and NP-EDF

In Table V and VI, we evaluate the flexibility of our framework using the scheduling policies originally targeted by the baseline algorithms (*i.e.*, P-EDF for *CaM*, NP-EDF for *IA*^{3} and *PDPA*).[5] We modified our framework to use the appropriate schedulability test—NP-EDF test [56] (used by [10], [11]), P-EDF test [57] (used by [5])—in the inner layer. The middle and outer layers have not been changed.

[5]Graphs for all results have been uploaded as supplemental material and will be made available as a technical report.

TABLE V: Total number of schedulable task sets in each scenario using different algorithms under NP-EDF. In **Bold** the results of the best-performing algorithm.

Scenario	*COMP*	*CASE*	*IA*^{3}	*PDPA*
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-B	**2096**	2089	2075	841
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-S1	**1695**	1684	1586	937
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-S2	1447	**1455**	1294	804
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-B	**2109**	1803	2075	767
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-S1	**1699**	1467	1572	688
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-S2	**1413**	1237	1272	572
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-B	2522	**2711**	2259	1514
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-S1	925	**972**	584	410
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-S2	677	**769**	213	139
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-B	**2519**	2237	2172	1350
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-S1	**914**	775	590	318
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-S2	**581**	501	192	106

Results for NP-EDF (Table V) show a similar trend as NP-FP (Table IV): *COMP* performs better than *IA*^{3} and *PDPA* for all scenarios, while *CASE* performs better than the baselines when tasks have higher cache sensitivities or shorter range of task periods. On average, *COMP* and *CASE* improve schedulability by 17.1% and 11.4%, respectively, over the best-performing baseline (*i.e.*, *IA*^{3}). For each scenario, if we consider our better-performing algorithm (*CASE* or *COMP*) and compare against the better-performing baseline, the average improvement is 19.2%. The maximum improvement of 261% is achieved by *CASE* in scenario $\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-S2.

Table VI reports the schedulability results under P-EDF, which was originally targeted by *CaM* [5]. The results show a different trend than under non-preemptive scheduling: *CASE* performs better than *CaM* in all scenarios and better than *COMP* in most cases. Unlike under non-preemptive scheduling, a longer-executing, lower-priority (or longer period) task cannot block a shorter-deadline higher-priority (or shorter period) task. Therefore, task periods and mutual compatibility play a less important role than cache sensitivity. Moreover, the overall improvement (8.7% on average) achieved by the proposed algorithms under P-EDF is smaller than NP-FP and NP-EDF. In fact, in *COMP* and *CASE*, the deciding factors (*i.e.*, cache sensitivity and task period) used in the middle layer are specifically proposed for non-preemptive tasks. As future work, we would like to investigate whether a new task selection mechanism for P-EDF could produce better results.

D. Running Time

We implemented all the algorithms under comparison using Python 3.10 and conducted our experiments on a Linux server equipped with Intel Xeon Gold 6254 CPU (3.10 GHz). Table VII presents the average and maximum running time (in seconds) required by the proposed optimization framework and the baseline algorithms. The running time does not include the time for deriving the execution time function for each task. For our experiments under NP-FP, *COMP* and *CASE* required average running time of less than 1 s and 2 s with 16 and 32 cache partitions, respectively, which are comparable with *IA*^{3} and faster than *CaM*. For the scenario under NP-EDF, the proposed framework ran slower than *IA*^{3} and *PDPA*, with infrequent spikes due to a combination of multiple iterations and the slower schedulability test. P-EDF running times are shorter than NP-FP and NP-EDF because of the faster utilization bound test.

TABLE VI: Total number of schedulable task sets in each scenario using different algorithms under P-EDF. In **Bold** the results of the best-performing algorithm.

Scenario	*COMP*	*CASE*	*CaM*
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-B	2096	**2099**	2071
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-S1	**1695**	1692	1663
$\mathbb{AR}$-I + $\mathbb{SH}$ + $\mathbb{SD}$-S2	1447	**1459**	1397
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-B	**2113**	2107	2074
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-S1	**1710**	1699	1675
$\mathbb{AR}$-I + $\mathbb{WD}$ + $\mathbb{SD}$-S2	1424	**1442**	1383
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-B	2522	**2711**	2617
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-S1	923	**977**	763
$\mathbb{AR}$-II + $\mathbb{SH}$ + $\mathbb{SD}$-S2	675	**770**	433
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-B	2519	**2660**	2578
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-S1	931	**1009**	773
$\mathbb{AR}$-II + $\mathbb{WD}$ + $\mathbb{SD}$-S2	641	**738**	406

TABLE VII: Running time comparison (avg/max) (in sec.).

Scenario	n_p	*COMP*	*CASE*	*IA*^{3}	*PDPA*	*CaM*
NP-FP	16	0.6/2.2	0.5/2.1	1.2/3.7	0.4/3.9	9.4/59
	32	2.0/9.6	1.5/7.4	1.8/7.7	0.8/8.7	22/135
NP-EDF	16	4.7/44	6.8/39	2.5/28	0.3/16	-
	32	16/190	19/168	4.0/68	0.5/18	-
P-EDF	16	0.2/1.0	0.2/0.8	-	-	2.9/13
	32	0.5/2.4	0.5/2.3	-	-	12/49

Furthermore, we also analyzed the running time scalability of the framework (for NP-FP) using hypothetical test cases with 16 cores, 128 cache partitions, and 160 tasks, which took an average running time of 22 mins. Note that our current implementation only utilizes one CPU core for one problem instance. Exploiting the inherent parallelism in the search in the outer layer can speed up the optimization manifolds.

VII. Conclusion and Future Work

Unlike in preemptive scheduling, a lower-priority non-preemptive task can *block* a higher priority task, significantly impacting the schedulability of a task set. Also, intuitively, clustering tasks with similar *cache sensitivities* to use a part of shared cache can maximize the cache utilization and improve the schedulability. This paper provides useful insights on the *trade-offs* between blocking and cache sensitivity in establishing the schedulability of a set of fixed-priority non-preemptive tasks on multi-core processors.

We propose a *multi-layer hybrid design space exploration framework* to solve the joint problem of cache partitioning and task allocation. Our extensive experimental evaluation against state-of-the-art algorithms shows that our framework can considerably improve real-time schedulability even when cache sensitivities of tasks cannot be fully exploited. Although some optimization strategies of the framework are specifically designed for NP-FP real-time tasks, we show that the framework achieves good schedulability results also for *preemptive* and *non-preemptive EDF* tasks.

While this paper evaluates blocking and cache sensitivity separately, we will investigate in the future how to combine these two factors into a metric that can drive task allocation. Besides, we can possibly determine sets of mutually-compatible tasks and then, for each set, perform task allocation based on cache sensitivity. Naturally, we will also consider complex task models (*e.g.*, directed acyclic graphs) and platform settings (*e.g.*, memory bandwidth regulation [36]).

REFERENCES

[1] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *ACM/IEEE Design Automation Conference (DAC)*, 2008.

[2] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[3] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[4] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[5] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee, "Holistic resource allocation for multicore real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[6] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller, "Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[7] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *IEEE Real-Time Systems Symposium (RTSS)*, 2010.

[8] R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems," *Real-Time Systems*, vol. 55, 2019.

[9] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real Time Systems*, vol. 35, p. 239–272, 2007.

[10] M. Paolieri, E. Quiñones, F. J. Cazorla, R. I. Davis, and M. Valero, "IA^3: An interference aware allocation algorithm for multicore hard real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[11] B. Berna and I. Puaut, "Pdpa: Period driven task and cache partitioning algorithm for multi-core systems," in *ACM Conference on Real-Time and Network Systems (RTNS)*, 2012.

[12] M.-K. Yoon, J.-E. Kim, and L. Sha, "Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[13] N. Suzuki, H. Kim, D. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *IEEE International Conference on Computational Science and Engineering (CSE)*, 2013.

[14] A. Sarkar, F. Mueller, and H. Ramaprasad, "Static task partitioning for locked caches in multicore real-time systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 1, 2015.

[15] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson, "Cache sharing and isolation tradeoffs in multicore mixed-criticality systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2015.

[16] N. Kim, B. C. Ward, M. Chisholm, C.-Y. Fu, J. H. Anderson, and F. D. Smith, "Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning," *Real-Time Systems*, vol. 53, pp. 709–759, 2017.

[17] J. Bakita, S. Ahmed, S. H. Osborne, S. Tang, J. Chen, F. D. Smith, and J. H. Anderson, "Simultaneous multithreading in mixed-criticality real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 278–291.

[18] G. Chen, B. Hu, K. Huang, A. Knoll, K. Huang, D. Liu, and T. Stefanov, "Automatic cache partitioning and time-triggered scheduling for real-time MPSoCs," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014.

[19] S. Altmeyer, R. I. Davis, and C. Maiza, "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *Real-Time Systems*, vol. 48, no. 5, pp. 499–526, 2012.

[20] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[21] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2002, pp. 114–123.

[22] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "On the effectiveness of cache partitioning in hard real-time systems," *Real Time Systems*, vol. 52, no. 5, pp. 598–643, 2016.

[23] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical os-level cache management in multi-core real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[24] H. Kim and R. Rajkumar, "Real-time cache management for multi-core virtualization," in *International Conference on Embedded Software (EMSOFT)*, 2016.

[25] J. Xiao, Y. Shen, and A. D. Pimentel, "Cache interference-aware task partitioning for non-preemptive real-time multi-core systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 21, no. 3, pp. 1–28, 2022.

[26] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen, "Dna: Dynamic resource allocation for soft real-time multicore systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 196–209.

[27] O. Kwon, G. Schwäricke, T. Kloda, D. Hoornaert, G. Gracioli, and M. Caccamo, "Flexible cache partitioning for multi-mode real-time systems," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021.

[28] R. Gifford and L. T. X. Phan, "Multi-mode on multi-core: Making the best of both worlds with omni," in *IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 118–131.

[29] Xilinx, "ZCU 102 MPSoC TRM," https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf, 2021.

[30] NVIDIA, "Xavier TRM," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[31] Intel, "Cache Allocation Technology," https://www.intel.com/content/www/us/en/developer/articles/technical/cache-allocation-technology-usage-models.html, 2016.

[32] ARM, "Arm Architecture Reference Manual Supplement. Memory System Resource Partitioning and Monitoring (MPAM) for Armv8-A," https://developer.arm.com/docs/ddi0598/.

[33] X. Pan and F. Mueller, "Controller-aware memory coloring for multicore real-time systems," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC)*, 2018, pp. 584–592.

[34] S.-W. Cheng, J.-J. Chen, J. Reineke, and T.-W. Kuo, "Memory bank partitioning for fixed-priority tasks in a multi-core system," in *IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 209–219.

[35] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 155–166.

[36] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[37] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange, "Run-time control to increase task parallelism in mixed-critical systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014, pp. 119–128.

[38] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011, pp. 269–279.

[39] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, and M. Caccamo, "Predictable Execution Model: Concept and Implementation," Tech. Rep., 2010. [Online]. Available: http://hdl.handle.net/2142/16605

[40] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable Flight Management System Implementation on a Multicore Processor," in *Embedded Real Time Software (ERTS)*, 2014.

[41] A. Melani, M. Bertogna, R. I. Davis, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Exact Response Time Analysis for Fixed Priority Memory-Processor Co-Scheduling," *IEEE Transactions on Computers (TC)*, vol. 66, no. 4, pp. 631–646, 2017.

[42] J. Arora, C. Maia, S. A. Rashid, G. Nelissen, and E. Tovar, "Schedulability analysis for 3-phase tasks with partitioned fixed-priority scheduling," *Journal of Systems Architecture (JSA)*, vol. 131, p. 102706, 2022.

[43] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[44] F. Dellaert and G. Contributors, "borglab/gtsam," May 2022. [Online]. Available: https://github.com/borglab/gtsam

[45] A. Vedaldi and B. Fulkerson, "VLFeat: An open and portable library of computer vision algorithms," http://www.vlfeat.org/, 2008.

[46] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[47] M. Moll, I. A. Şucan, and L. E. Kavraki, "Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization," *IEEE Robotics & Automation Magazine*, vol. 22, no. 3, pp. 96–102, September 2015.

[48] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: An open toolbox for adaptive wcet analysis," in *Software Technologies for Embedded and Ubiquitous Systems*, 2010, pp. 35–46.

[49] C. Ferdinand and R. Heckmann, "aiT: Worst-case execution time prediction by static program analysis," in *Building the Information Society*, 2004, pp. 377–383.

[50] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.

[51] G. Bernat, A. Colin, and S. Petters, "Wcet analysis of probabilistic hard real-time systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2002, pp. 279–288.

[52] B. Lesage, D. Griffin, F. Soboczenski, I. Bate, and R. I. Davis, "A framework for the evaluation of measurement-based timing analyses," in *ACM International Conference on Real Time and Networks Systems (RTNS)*, 2015.

[53] F. Farshchi, P. K. Valsan, R. Mancuso, and H. Yun, "Deterministic memory abstraction and supporting multicore system architecture," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[54] S. Roozkhosh and R. Mancuso, "The potential of programmable logic in the middle: Cache bleaching," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[55] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS at the Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[56] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *IEEE real-time systems symposium (RTSS)*, 1991, pp. 129–139.

[57] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.