Minimizing Cache Usage for Real-time Systems

Binqi Sun, Tomasz Kloda, Sergio Arribas Garcia, Giovani Gracioli, Marco Caccamo

Minimizing Cache Usage for Real-time Systems

Binqi Sun
Technical University of Munich
Munich, Germany
binqi.sun@tum.de

Tomasz Kloda
LAAS-CNRS, Université de Toulouse, INSA
Toulouse, France
tkloda@laas.fr

Sergio Arribas Garcia
Federal University of Santa Catarina
Florianópolis, Brazil
sergio.arribas.garcia@posgrad.ufsc.br

Giovani Gracioli
Federal University of Santa Catarina
Florianópolis, Brazil
giovani@lisha.ufsc.br

Marco Caccamo
Technical University of Munich
Munich, Germany
mcaccamo@tum.de

ABSTRACT

Cache partitioning is a technique to reduce interference among tasks accessing the shared caches. To make this technique effective, cache segments must be given to the tasks that can benefit most from having their data and instructions cached for faster execution. The existing partitioning schemes for real-time systems divide the available cache among the tasks to guarantee their schedulability which is the sole optimization criterion. However, it is also preferable, especially in systems with power constraints or mixed criticalities, to reduce the total cache usage for real-time tasks.

In this paper, we develop optimization algorithms for cache partitioning that, besides ensuring schedulability, also minimize cache usage. We consider both preemptive and non-preemptive scheduling policies on single-processor systems. For preemptive scheduling, we formulate the problem as an integer quadratically constrained program and propose an efficient heuristic achieving near-optimal solutions. For non-preemptive scheduling, we combine linear and binary search techniques with different schedulability tests. Our experiments based on synthetic task sets with parameters from real-world embedded applications show that the proposed heuristic: (i) achieves an average optimality gap of 0.79% within 0.1x run time of a mathematical programming solver and (ii) reduces average cache usage by 39.15% compared to existing cache partitioning approaches. Besides, we find that for large task sets with high utilization, non-preemptive scheduling can use less cache than preemptive to guarantee schedulability.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**.

KEYWORDS

cache partitioning, real-time, optimization, local search

RTNS 2023, June 7–8, 2023, Dortmund, Germany

ACM ISBN 978-1-4503-9983-8/23/06…$15.00
https://doi.org/10.1145/3575757.3593651

ACM Reference Format:
Binqi Sun, Tomasz Kloda, Sergio Arribas Garcia, Giovani Gracioli, and Marco Caccamo. 2023. Minimizing Cache Usage for Real-time Systems. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 7–8, 2023, Dortmund, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3575757.3593651

1 INTRODUCTION

The main benefit of cache partitioning in real-time systems is that it removes inter-task interference: preempting task will not evict the cached memory blocks of preempted task if both tasks use separate cache partitions. Cache partitioning can be implemented using specific hardware extensions (*e.g.,* Intel's Cache Allocation Technology [33] or ARM's Lockdown by master [43]) [23] or in software by exploiting address mapping between main memory and cache lines (*e.g.,* cache coloring) [35, 38, 45, 60]. However, if the task's working set does not fit into the task's private cache partition, the task will see an increased number of cache misses and, consequently, increased execution time. To mitigate this problem, various optimization techniques are used to allocate cache partitions of adequate size to tasks in function of their timing constraints.

Cache partitioning optimization methods for real-time systems focus on finding the cache partitioning under the assumption that all available cache segments can be allocated to the tasks. Despite the wealth of the literature, reducing cache usage is not part of the optimization criteria. However, for a variety of reasons, unrestrained cache usage might be of concern to embedded engineers. Multilevel caches often consume about half the processor energy [32], and choosing the processors with a last-level cache size fitting the application requirements or appropriately selecting its size can largely reduce power dissipation. Otherwise, the unallocated partitions can be used to improve the quality of service of the best-effort tasks. In the context of partitioned multi-core systems where the task-to-core allocation is fixed beforehand, the cache partitioning problem boils down, in fact, to minimize the cache usage of every core while ensuring its schedulability (see Example 1.1). When a task-to-core allocation is not given, the cache minimization can be used as a sub-procedure in task and cache co-allocation method.

Example 1.1. Consider a multicore mixed-criticality system from Figure 1 where three real-time operating systems ($RTOS_1$, $RTOS_2$, and $RTOS_3$) are running alongside a general purpose operating system ($GPOS_1$). Each operating system runs a set of tasks on a dedicated core. The tasks are statically allocated to the cores and

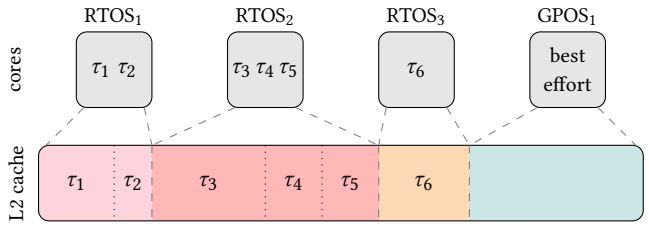


Figure 1: Last-level cache dimensioning in a multicore mixed-criticality system.

do not migrate from one core to another due to certification restrictions (*e.g.,* ensuring isolation among tasks with different criticality levels [21]) or interoperability issues (*e.g.,* missing libraries). The processor has an L2 cache shared among all cores and operating systems reducing the performance gap between the processor and main memory. Cache partitioning is used to avoid inter-core interference (*i.e.,* tasks running on different cores access the shared cache simultaneously and evict their cached blocks) and intra-core interference (*i.e.,* preempting task evicts the cached blocks from the preempted tasks on the same core). Finding a minimal size of cache partition for each real-time operating system is crucial for determining a maximal size of cache partition that can be used by the general-purpose operating system. The partitioning method can either assign cache partitions on a per-task basis (each task can have a private L2 cache partition, *e.g.,* [35]) or on a per-core basis (each core can have a private L2 cache partition but the tasks running on the same core cannot be assigned different sub-partitions, *e.g.,* [38]). In the former case, we propose a cache minimization for single-core preemptive systems where each task can have a private cache partition, and in the latter case for non-preemptive systems where all tasks execute non-preemptively using the same cache partition without incurring cache-related preemption delays.

An apparent solution at hand for minimizing the cache usage is to invoke iteratively one of the standard cache partitioning methods with a smaller (bigger) cache size at each step until the system becomes unschedulable (schedulable). Indeed, several cache allocation methods are easily amenable to this approach or can stop earlier when schedulability is guaranteed, and there is no point in further reduction of the system utilization (*e.g.,* gradient descent for minimizing system utilization [36] discussed in Section 5.1.4). In this research, we also report such methods and describe the required modifications. On the other hand, some methods use remaining cache segments to allow faster convergence (*e.g.,* branch-and-bound [4]), and specific approaches must be proposed. Moreover, restarting the search for each cache size without any knowledge of the previous iterations might not be particularly efficient, and certain proprieties of the schedulability tests, in particular, sustainability as suggested in [4], can be exploited to skip the redundant tests when going from one cache partition size to another.

This paper makes the following new contributions. For single-core preemptive scheduling, we formulate the cache minimization problem as an integer quadratically constrained program (IQCP), which can be solved optimally by a standard mathematical programming solver. To improve the efficiency of the IQCP solution, we propose a guided local search (GLS) heuristic that can obtain near-optimal solutions in a fraction of the solver's run time. Moreover, we apply the branch-and-bound (BB) and dynamic programming (DP) methods to the cache minimization problem. For single-core non-preemptive scheduling, we derive pseudo- and fully-polynomial time search algorithms that incorporate different schedulability tests. To evaluate the proposed methods, we conduct simulation experiments based on embedded programs to quantitatively compare different approaches in terms of their cache usage, schedulability ratio, and run time.

2 CACHE PARTITIONING METHODS

Cache partitioning is a technique to assign portions of the cache to either tasks or cores to reduce interference (both intra- or inter-core) and increase the predictability of the system. There are two ways of performing cache partitioning in modern processors: (i) *index-based*, where partitions are formed by an aggregation of associative sets in the cache; or (ii) *way-based*, where partitions are formed by an aggregation of individual cache ways [23].

Index-based cache partitioning can be implemented using specific hardware extensions [52] or by software within the operating system by relying on specific processor features, such as virtual memory (to implement cache coloring for instance) [24, 35, 38]. Way-based techniques have the advantages of not demanding changes in the cache organization and to isolate the requests for the different partitions from each other (no contention for cache ways in the cores). However, an important drawback of the way-based methods is the limited number of partitions and granularity of allocations due to the associativity of the cache [23].

For index-based methods implemented in software, such as cache coloring, there are also two implementation choices: (i) to assign partitions to tasks; or (ii) to assign partitions to cores. The former demands the operating system to be aware of the cache partitions and somehow implement the assignment of partitions to tasks in its memory allocator [24]. The latter is very useful, for instance, in hypervisor-based systems, where the hypervisor is responsible for assigning partitions to cores, despite the operating systems and the number of tasks running on top of it, providing cache partitioning to operating systems that do not support it [25]. The main advantage of assigning partitions to tasks is the possibility of having the best match between the cache space (*i.e.,* individual cache partition sizes) and the worst-case execution times of tasks.

We note that several changes to the microarchitecture have been proposed for more flexible cache partitioning in real-time and mixed-criticality systems [22, 42].

3 RELATED WORK

We review different cache partitioning techniques and discuss how these techniques can be adapted to minimize cache usage in real-time systems. In one of the earliest works, Kirk [36] attempts to minimize task set utilization by allocating the cache segments to the tasks for which it leads to the highest change in the total utilization (*i.e., gradient descent*). Although the main objective is to minimize the utilization, the method can be easily modified to stop when the schedulability is guaranteed by the utilization bound. Plazar et

al. [49] formulate an *integer linear problem* to allocate cache partitions minimizing the total length of tasks' worst-case execution times that can be easily replaced with utilization. Sasinowski and Strosnider [50] apply *dynamic programming* to minimize task set utilization. The algorithm adds tasks one by one, looking for the cache allocation minimizing the utilization of the current subset of tasks for every possible number of segments. This is done by combining the previous allocations with the allocations of the task that has just been added. (*e.g.,* the minimal utilization for six tasks using two segments can be found as the sum of minimal utilizations of five previous tasks using, respectively, zero, one or two segments and the sixth task using two, one or zero segments, respectively). Considering every possible number of segments is not efficient for finding the minimal cache usage. However, the algorithm can be modified for this purpose as shown in Section 5.1.4. Bui et al. [12] propose a *genetic algorithm* to solve the cache to task allocation problem. While all the methods mentioned above consider utilization as minimization criteria, Altmeyer et al. [4] show that this might not be optimal with respect to schedulability. To find the optimal cache partitioning in this respect, the authors propose a *branch-and-bound* search combined with exact response time analysis. The proposed search technique cannot be directly applied to minimize the cache usage as at each step it tries to allocate all remaining cache segments in order to accelerate the convergence. We report the required modifications in Section 5.1.3.

An alternative approach to the problem of shared cache is to allow the tasks to use the entire cache and take into account the *cache-related preemption delays* (*CRPD*) [3, 7]. Minimizing the cache usage for such systems would involve a different optimization problem [27, 28] where the inter-task cache interference can be characterized using the concept of an interference matrix. This approach may lead to better schedulability performance and more efficient cache usage. However, adding cache effect to schedulability analysis [1, 2, 14, 15, 55, 63, 64] is not widely supported by the static program analysis tools. For instance, *aiT WCET Analyzer* supports this feature only for three targets, namely, *LEON2*, *e200*, and *e300* [1].

While our work targets single-core, the *cache-cognizant scheduling policies* have been proposed for global multi-core systems. These policies dispatch a new task for execution if there is an idle processor and a sufficient number of cache segments (each task has a constant and predefined cache requirement) [26]. Other cache-aware scheduling policies can promote the execution of tasks sharing a common working set [16] or preempt the tasks running on the remote cores using the same cache partition as the preempting task [56]. For cache-agnostic global schedulers, Xiao et al. [57] propose the schedulability analysis that accounts for cache interference. In multi-core systems, other shared resources, like memory bandwidth, may also degrade the system's predictability. Several works propose coordinated cache and bandwidth co-allocation [47, 53, 59]. Although we do not consider memory bandwidth in our analysis, different solutions can be used alongside to mitigate the interference due to DRAM bank sharing (*e.g.,* bank partitioning [17, 46, 61], software bandwidth regulators [39, 62] or segmented execution models [20, 48]). If the task-to-core mapping is not given as assumed in this work, various task and cache co-allocation methods can be applied [27, 58, 59].

[1] https://www.absint.com/ait/ucb_analysis.htm

4 SYSTEM MODEL

We consider a system with n tasks scheduled by a fixed-priority preemptive or non-preemptive scheduler on a single-core platform. Additionally, under preemptive scheduling, each task can be assigned a private cache partition, and under non-preemptive one, all the tasks share one single cache partition.

The cache has a size of S and is divided into m equally-sized separate segments. Each task can be assigned an arbitrary number of cache segments for its individual use. The set of segments owned by a task is its partition [50]. Under preemptive scheduling policy, the cache partitions cannot be shared among different tasks as it might lead to inter-task cache eviction resulting in cache-related preemption delays. Obviously, this constraint is not necessary under non-preemptive policies, and we can consider that all tasks share all assigned segments. If task τ_i is assigned k cache segments, its worst-case execution time (*WCET*, *i.e.,* the longest task execution time when running stand-alone) is given by $C_{i,k}$. Additionally, we denote by $C_{i,0}$ task τ_i's worst-case execution time with zero cache partition. We assume the monotonicity of the execution times with respect to the number of cache segments: by assigning more cache segments to the tasks, their worst-case execution time will not increase. We acknowledge that the execution times might not be necessarily monotonic, but the impact of these effects is limited [4], and smaller partitions can be used instead. Several previous works consider similar memory model [4, 12, 37, 49].

Each task τ_i gives rise to a potentially infinite sequence of identical jobs (instances) released sporadically after the minimum inter-arrival time or period T_i. Each job released by task τ_i is characterized by a relative deadline D_i assumed to be less than or equal to the task period (*i.e., constrained* deadlines): $D_i \leq T_i$, and its worst-case execution time that depends on the number of cache segments assigned to the task. All the above parameters are positive integers. We will also use $U_{i,k} = C_{i,k}/T_i$ to refer to the task τ_i utilization when executing with k cache segments at its disposal. Additionally, we use $U = \sum_{i=1}^{n} U_{i,0}$ to denote the task set base utilization (*i.e.*, the sum of task utilization without using cache).

The tasks are scheduled by a fixed-priority scheduler, and each task is assigned a unique priority. Tasks are indexed in decreasing priority order (τ_1 has the highest priority and τ_n has the lowest priority). In this work, in particular, we assume *Rate Monotonic* (*RM*) [44] assignment rule where task priorities are inversely proportional to task periods. The worst-case response time R_i of task τ_i is defined as the longest time from the release of a job of the task until its completion. If the worst-case response time is less than or equal to the task τ_i deadline ($R_i \leq D_i$), we say that task τ_i is *schedulable*. Likewise, we say that a task set is schedulable if all its tasks are schedulable.

5 MINIMIZING CACHE USAGE

We look for a cache-to-task assignment for which all n tasks are schedulable, and a minimal number of cache segments is used. We consider preemptive and non-preemptive scheduling.

5.1 Preemptive scheduling

The cache partitioning problem for preemptive scheduling can be shown NP-hard by reduction to the knapsack problem [12]. We propose four methods to minimize cache usage under preemptive scheduling. To verify the schedulability, we will use utilization bound test [44] with linear and response time analysis (RTA) [5, 34] with pseudo-polynomial time complexity in the number of tasks.

The worst-case response time R_i of task τ_i can be computed by a fixed-point iteration of the following formula:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{1}$$

5.1.1 *Mathematical programming.*

For each task τ_i with $i = 1, \ldots, n$, we define a variable C_i that represents its worst-case execution time. For each $k = 0, 1, \ldots, m$, we introduce the binary variables $x_{i,k}$ which take on value 1 if and only if task τ_i is assigned k cache segments and 0 otherwise:

$$\forall i = 1, \ldots, n : \; C_i = \sum_{k=0}^{m} x_{i,k} \cdot C_{i,k} \tag{2}$$

We additionally require that each task τ_i has exactly one variable $x_{i,k}$ for all $k = 0, 1, \ldots, m$ that is equal to 1:

$$\forall i = 1, \ldots, n : \; \sum_{k=0}^{m} x_{i,k} = 1 \tag{3}$$

Our objective function is to minimize the total cache usage:

$$\textbf{minimize} \sum_{i=1}^{n} \sum_{k=0}^{m} x_{i,k} \cdot k \tag{4}$$

Now, we add the constraints to ensure the schedulability of all tasks using the response time analysis formulation proposed by Baruah and Ekberg [6] and by Davare et al. [18]. We introduce $\hat{R}_i$ as the upper bound of task τ_i's worst-case response time (*i.e.*, $\hat{R}_i \geq R_i$). All tasks are schedulable if we can find such values of $\hat{R}_i$ for all $i = 1, \ldots, n$ that are less than or equal to their respective deadlines:

$$\forall i = 1, \ldots, n : \hat{R}_i \leq D_i \tag{5}$$

The variable $\hat{R}_i$ upper bounds the task τ_i's worst-case response time if the following constraints are satisfied:

$$\forall i = 1, \ldots, n : \; C_i + \sum_{j=1}^{i-1} Z_{i,j} \cdot C_j \leq \hat{R}_i \tag{6}$$

where $Z_{i,j}$ is an integer that upper bounds the number of times a higher-priority task τ_j can preempt task τ_i during its worst-case response time, and thus, it must satisfy another constraint:

$$\forall i = 1, \ldots, n, j = 1, \ldots, i-1 : \; Z_{i,j} \cdot T_j \geq \hat{R}_i \tag{7}$$

Since Formula (6) is a quadratic constraint, the resulting mathematical model is an integer quadratically constrained program (IQCP), where the objective function is formulated by Formula (4), and the constraints are formulated by Formulas (2), (3) and (5-7).

The IQCP can be solved by a standard mathematical programming solver. However, its worst-case run time is exponential in relation to the task set size. To address this, we propose an alternative approach in subsection 5.1.2 called guided local search (GLS), which is a simple and efficient heuristic method that can significantly reduce the run time complexity.

5.1.2 *Guided local search.*

The GLS proposed in this paper is an iterative local search algorithm that utilizes problem-specific knowledge to guide the search direction and tabu search mechanism to avoid revisiting duplicate solutions. The procedures of GLS are outlined in Algorithm 1.

The search starts from an initial solution s, where all the tasks are allocated with the maximal number of segments m (line 1). If the initial solution is not schedulable, we stop the algorithm since the task set cannot be feasible with fewer cache segments (lines 2-3). Otherwise, we continue the algorithm to do the iterative search (lines 5-10). The iterative search is divided into two phases (*i.e., decrease phase* and *increase phase*) depending on the schedulability of the current solution. In the decrease phase (lines 5-6), the current solution s is schedulable, and in each step, one task is selected to decrease its cache partition. As a result, the solution moves from schedulable to unschedulable gradually. Once the solution crosses the schedulability boundary, the algorithm enters the increase phase (lines 7-8), where the current solution increases its cache partition, moving back to schedulable. Whenever the algorithm visits a new solution, its schedulability is checked by the RTA, and the best-so-far solution s^* will be updated by s if the latter uses less cache and is schedulable (line 9). The algorithm terminates when the number of schedulability checks l reaches its upper limit L, which is a parameter defined by the user.

Algorithm 1: Guided local search

Input: Set of n tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, available cache segments m, upper limit of schedulability test invocations L;
Output: Best cache allocation $s^* = \{s_1, ..., s_n\}$;

```
1  Initialize s ← {m, ..., m}, l ← 1;
2  if s is not schedulable then
3      return false;
4  while l ≤ L do
5      if s is schedulable then                    // decrease phase
6          Select a task to decrease its cache partition in s;
7      else                                        // increase phase
8          Select a task to increase its cache partition in s;
9      Check the schedulability of the current solution s and update
         the best-so-far solution s*;
10     l ← l + 1;
11 return s*;
```

Now, we explain the details of the neighborhood structure, task selection rule, and the tabu mechanism used in the GLS.

- **Neighborhood structure** defines the solutions that can be reached by the current solution in one local search step. In the decrease (increase) phase of the GLS, the neighborhood structure is defined as the n solutions, in each of which one task is selected to decrease (increase) its cache partition to the least number of segments that increases (decreases) the task's WCET by one step. For clarity, we illustrate this with an example. Suppose we have a task set consisting of two tasks `pca` (τ_1) and `stitch` (τ_2), whose

benchmarked WCET profiles are given in Figure 3. The current cache partition of the two tasks is {1024, 512} KB, and we assume it to be schedulable. Then, the two neighborhood solutions obtained by decreasing the cache partitions of task τ_1 and τ_2 are {512, 512} and {1024, 256}, respectively.

- **Task selection rule** determines which task is selected to decrease (increase) its cache partition and move the solution to the corresponding neighborhood. In the decrease (increase) phase, we select the task with the maximal (minimal) value of $\delta_i = \downarrow s_i / \uparrow U_i$ ($\delta_i = \uparrow s_i / \downarrow U_i$), where $\downarrow s_i$ ($\uparrow s_i$) denotes task τ_i's cache decrease (increase) size and $\uparrow U_i$ ($\downarrow U_i$) denotes the resulted utilization increase (decrease). The intuition behind it is to decrease the most (increase the least) cache usage with the least increase (most decrease) of total utilization. We use the same example task set as before to illustrate the task selection rule. Suppose the utilization increase of the two tasks are $\uparrow U_1$=0.5 and $\uparrow U_2$=0.2. The decrease of their cache partitions can be calculated as $\downarrow s_1$=1024−512=512 and $\downarrow s_2$=512−256=256. Then, we have δ_1=512/0.5=1024 and δ_2=256/0.2=1280. In this case, we will select task τ_2 to decrease its cache partition since $\delta_2 > \delta_1$.
- **Tabu mechanism** is used to avoid the search visiting duplicated solutions. In the GLS, whenever a new solution is visited, we use a hash function to map the solution into an integer and save it to a *visit history*. Meanwhile, at each step of the local search, we move the current solution to a neighborhood only if it has not been visited in the history. If all the neighborhood solutions have already been visited, we restart the search by re-initializing the current solution as a random solution to escape the local optima.

5.1.3 *Branch-and-bound.*

We propose a branch-and-bound (B&B) algorithm inspired by [4]. Our branching strategy consists of allocating the cache to one task at one time. The initial node of the B&B generates all possible cache partitions for the first task, creating a branch for each partition. The algorithm then expands these branches by generating all possible cache partitions for the current task in the current node. A solution is considered valid when the cache partition of all tasks has been specified and the resulting task set is schedulable.

To reduce the number of partial solutions to be explored, we propose a pruning strategy that checks the schedulability of each partial solution under an optimistic assumption that each not yet specified task partition equals the current remaining cache segments. If the task set is not schedulable under this assumption, the current partial solution is pruned. Additionally, once a valid solution is found, we update the available cache to be one segment lower than the best-so-far cache usage m^* to further improve the pruning efficiency. The algorithm terminates if an optimal solution has been found or an upper limit of schedulability test invocations L has been reached.

The property of the *sustainability* [13] of the schedulability test together with the assumption the execution times are monotonically non-increasing with the number of cache segments is exploited to reduce the number of test invocations [4]. If tasks from τ_1 to τ_{i-1} are deemed schedulable under cache allocation $s^{(i-1)}$, then they will remain schedulable under any cache allocation with greater or equal cache partitions. Moreover, as tasks τ_1-τ_{i-1} response times do not depend on lower-priority tasks τ_i-τ_n, tasks τ_1-τ_{i-1} do not have to be tested when branching adds a new partition for task τ_i.

Algorithm 2: Cache usage minimization using B&B

Input: Set of n tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, available cache segments m, upper limit of schedulability test invocations L;
Output: Best cache allocation $s^* = \{s_1, ..., s_n\}$, minimal number of cache segments m^* needed by τ;

```
1  Function minimize_cache(τ, s^(i−1), s*, m*, L):
2      if the number of schedulability tests reaches upper limit L then
3          return s*, m*;
4      if i − 1 = n then
5          if τ is schedulable with s^(n) and ∑ s^(n) < m* then
6              s* ← s^(n); m* ← ∑ s*;
7          return s*, m*;
8      m′ ← m* − ∑ s^(i−1) − 1;
9      if τ is not schedulable with {s_1, ..., s_i = m′, ..., s_n = m′} then
10         return s*, m*;
11     s_i ← 0;
12     while s_i ≤ m′ do
13         s^(i) ← {s_1, ..., s_{i−1}, s_i};
14         s*, m* ← minimize_cache(τ, s^(i), s*, m*, L);
15         s_i ← next_corner_point(τ_i);
16         m′ ← m* − ∑ s^(i−1) − 1;
17     return s*, m*;
18 s* ← {s_1 = m, ..., s_n = m}; s^(0) ← ∅;
19 return minimize_cache(τ, s^(0), s*, m + 1, L);
```

The detailed procedures of the proposed B&B can be found in Algorithm 2. It defines a recursive function `minimize_cache`, which takes a partial solution $s^{(i-1)}$ containing the cache partition of the first $i-1$ tasks as input and outputs the best-so-far solution s^* with its cache usage m^*. Lines 2-3 check whether the upper limit of the schedulability test invocations has been reached. Lines 4-7 check if the current solution is a complete solution containing the cache partitioning of all tasks. For a complete solution, it checks its schedulability and updates the best-so-far cache usage if it is schedulable. In line 6, $\sum s^{(n)}$ computes the total number of cache segments used by solution $s^{(n)}$. Line 8 updates the number of remaining cache segments m' according to the current best-so-far solution. Lines 9-10 implement the pruning strategy to discard the partial solution if it cannot be schedulable even with the optimistic assumption. Lines 11-15 implement the branching strategy starting from the first task with the least cache partition. Function `next_corner_point`(τ_i) gives the next cache partition size for which the worst-case execution time of task τ_i drops (see Example 5.1). In lines 17-18, the algorithm initializes $s^*, s^{(0)}$ and invokes the recursive function to minimize cache usage.

Example 5.1. Figure 2 shows the execution time of *sift-vga* sample program from the CortexSuite benchmarks [54] (see Section 6.1 for more details) as a function of cache size partition size from 0 to 512 KB with a step of 32 KB. The original curve (blue color) is comprised of 17 points. It can be easily seen that many of these points are redundant. Indeed, the execution time does not decrease

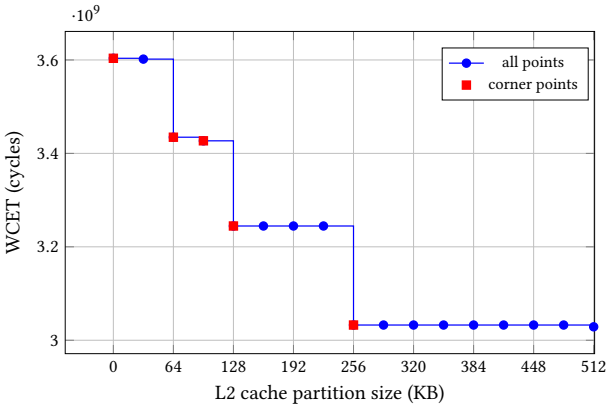


Figure 2: Execution time vs cache size for sift-vga.

Algorithm 3: Cache usage minimization using DP

Input: Set of n tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, available cache segments m;
Output: Best cache allocation $s^* = \{s_1, ..., s_n\}$;

```
for k = 0 to m do
    for i = 1 to n do
        P_{i,k} ← arg min_{0≤s_i≤k} (U_{i,s_i} + M_{i-1,k-s_i});
        M_{i,k} ← U_{i,P_{i,k}} + M_{i-1,k-P_{i,k}};
    if M_{n,k} ≤ U(n) then                          // schedulable
        m' ← k;
        for i = n to 1 do                           // retrieve cache allocation
            s_i ← P_{i,m'};
            m' ← m' − P_{i,m'};
        return s* = {s_1, ..., s_n};
return unschedulable;
```

continuously, but stays on a certain level and then makes a sudden jump down as the cache partition becomes large enough to hold the next important data structure [8]. The corner points leading to a decrease in execution time are marked in red in Figure 2.

5.1.4 *Dynamic programming.*

We present a dynamic programming (DP) based on [50] to minimize cache usage. Similar to [50], we define $M_{i,k}$ as the minimum utilization of tasks $\tau_1, ..., \tau_i$ if there are k cache segments available to them:

$$M_{i,k} = \min_{0 \le s_i \le k} \sum_{j=1}^{i} U_{j,s_j} \tag{8}$$

where s_i denotes the number of cache segments allocated to task τ_i. We also define $P_{i,k}$ to be the number of segments to allocate to τ_i to get $M_{i,k}$. The key observation in [50] is the following recurrence relation:

$$M_{i,k} = \min_{0 \le s_i \le k} (U_{i,s_i} + M_{i-1,k-s_i}) \tag{9}$$

which provides a simple way to compute $M_{i,k}$ for all tasks and cache sizes in an iterative manner starting from $i = 1, k = 0$ towards $i = n, k = m$. However, unlike [50], our algorithm does not need to enumerate all the $M_{i,k}$ values since it is designed to minimize cache usage while ensuring schedulability. For this purpose, we present Algorithm 2, which includes an early stopping criterion in lines 5-10. The early stopping criterion checks the schedulability using utilization bound $U(n) = n(2^{1/n} - 1)$ [44] and outputs the first cache partitioning solution that results in a schedulable task set. Additionally, we have reversed the inner and outer loop for faster convergence.

5.2 Non-preemptive scheduling

We apply two basic search strategies, *linear* and *binary search*, to find the minimal number of cache segments that guarantee the system schedulability under fixed-priority non-preemptive policy. The advantage of non-preemptive policy is that the tasks executing non-preemptively can share the same partition without any interference. Hence, the problem is less complex than in the fully preemptive scheduling as there are only $m + 1$ different cache partitionings to consider.

We recap the response time analysis for non-preemptive fixed-priority scheduling [10, 19]. In *non-preemptive* scheduling, every job executes from its start uninterruptedly until completion. The *priority-inversion* might happen when a higher-priority job must wait for the completion of a lower-priority job released before. The longest lower-priority *blocking* time that task τ_i can experience is given by the longest worst-case execution time among all low-priority tasks:

$$B_i = \max_{j>i} C_j \tag{10}$$

The schedulability analysis of fixed-priority non-preemptive systems requires checking multiple jobs of the same task [19]. It might be the case that the second, third, or later job has larger response time than the first job. This anomaly is known as *self-pushing*: while the job under analysis is executing, it blocks all higher-priority jobs released after the start of its execution; consequently, the next job will experience their accumulated interference. Therefore, the analysis of task τ_i must check all its jobs released within i-level *busy-period* defined as the longest time interval during which the jobs with priorities equal to i or higher are pending:

$$L_i = B_i + \sum_{j=1}^{i} \left\lceil \frac{L_i}{T_j} \right\rceil \cdot C_j \tag{11}$$

The above relation can be solved by fixed-point iteration. We should check the schedulability of each task τ_i instance within time interval $[0, L_i]$. The l-th job of task τ_i can start its execution when there are no other pending tasks. Its starting time must fulfill the following relation that can be solved by fixed-point iteration:

$$s_{i,l} = B_i + (l-1) \cdot C_i + \sum_{j=1}^{i-1} \left\lceil \frac{s_{i,l}}{T_j} \right\rceil \cdot C_j \tag{12}$$

The worst-case response time $R_{i,l}$ of l-th job of task τ_i is C_i time units after its start:

$$R_{i,l} = s_{i,l} + C_i \tag{13}$$

The task τ_i worst-case response time R_i is the maximum worst-case response time of its jobs:

$$R_i = \max_l R_{i,l} \tag{14}$$

5.2.1 Linear search.

In linear search, we check every number of cache segments in ascending order until we find one for which all tasks are schedulable. We apply the schedulability test for each task in decreasing priority order. If the test fails for a given number of cache segments, we assign an additional segment and repeat the test. However, it is not necessary to resume the schedulability test from the first task if the test is *sustainable* with respect to the WCETs [4]. All tasks that were previously proved schedulable will also be schedulable with one more cache segment. Their response times depend on the task WCETs, which by adding more cache segments, will decrease or remain unchanged. Hence, the response times cannot increase, and each task deemed schedulable with a given number of cache segments will remain schedulable with more cache segments. In the worst case, the linear search method invokes $n + m$ times the schedulability tests. Algorithm 4 shows the linear search approach for cache minimization.

Algorithm 4: Cache usage minimization using linear search

Input: Set of n tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, available cache segments m;
Output: Minimal number of cache segments k needed by τ;

```
Initialize k ← 0, i ← 1;
while k ≤ m do
    while τ_i is schedulable with k segments and i ≤ n do
        i ← i + 1;
    if i = n + 1 then
        return k;
    k ← k + 1;
return unschedulable;
```

We can further improve the cache minimization procedure by observing that the tasks' execution times do not decrease at every cache segment (see Example 5.1). If task τ_i is not schedulable with k cache segments under non-preemptive scheduling, and i) its worst-case execution time, ii) worst-case execution time of all higher-priority interfering tasks, and iii) maximal worst-case execution time among all lower-priority tasks are the same with $k + 1$ cache segments as with k cache segments, then task τ_i cannot be made schedulable on $k + 1$ cache segments. This means that we can skip the schedulability test for τ_i with $k + 1$ cache segments and keep increasing the size of the cache partition as long as there is no change in any worst-case execution time of the previously mentioned tasks.

5.2.2 Binary search.

The second strategy is based on the binary search. For each single task, we look for a minimum number of cache segments ensuring its schedulability. This number cannot be less than any minimum number of segments for which previously tested tasks were schedulable. The method starts with half of the available cache segments and checks the first task schedulability. In case of the test success (failure), the segments' numbers greater (less) than the tested number are discarded. The search continues on the remaining number of cache segments from its middle number and repeats the same procedure until the number of segments cannot be decreased anymore. Then, the search is applied for the next task in the decreasing priority order. Like in the linear search, we do not need to test again the tasks that were already deemed schedulable if the test is sustainable with respect to the worst-case execution times. We do neither consider the segments' numbers that were too small to ensure the schedulability of the previous tasks. The total number of schedulability test invocations is upper bounded by $n \log m$. Algorithm 5 outlines the binary search-based cache minimization.

Algorithm 5: Cache usage minimization using binary search

Input: Set of n tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, available cache segments m;
Output: Minimal number of cache segments k_{min} needed by τ;

```
Initialize k_min ← 0, i ← 1;
while i ≤ n do
    k_max ← m;
    while k_min ≤ k_max do
        k ← ⌊(k_min + k_max)/2⌋;
        if τ_i is schedulable with k segments then
            k_max ← k − 1;
        else
            if k_min = m then
                return unschedulable;
            else
                k_min ← k + 1;
    i ← i + 1;
return k_min;
```

The overall search time complexity depends on the schedulability test. The response time analysis (NP-RTA) giving sufficient and necessary schedulability conditions [19] has pseudo-polynomial time complexity. The fully polynomial complexity can be achieved by sufficient but not necessary schedulability tests. For instance, we can test the schedulability of each task with one single problem interval of length $D_i - C_i$ (Equation (16) in [19], denoted by NP-SINGLE) and upper bound task τ_i's latest starting $\hat{s}_i$:

$$\hat{s}_i = \max\{B_i, C_i\} + \sum_{j=1}^{i-1} \left\lceil \frac{D_i - C_i}{T_j} \right\rceil \cdot C_j \tag{15}$$

If $\hat{s}_i \leq D_i - C_i$, we can conclude that any job of task τ_i can start early enough to finish its execution before its deadline. Another approach worth considering is the use of the hyperbolic utilization bound (NP-HYPER) derived by Theorem 9 in [11]. Note that while the first two tests can skip tasks that are already deemed schedulable, the utilization bound must be recalculated by adding utilization factors of all tasks when increasing the cache size.

6 EVALUATION

6.1 Benchmarks and task sets

To evaluate the proposed approaches to minimize cache usage, we first performed experiments on benchmark applications executing under different cache partition sizes. We used *Cachegrind* [51], which is an open-source cache profiler tool, to run the benchmarks and collect cache-related values (*e.g.*, cache miss ratio). We modified *Cachegrind* to support arbitrary cache sizes (by default, it must be power of two). While the values obtained from *Cachegrind* cannot be interpreted as worst-case execution time upper bounds, the tool helps us understand the relation between partition size and execution speed.

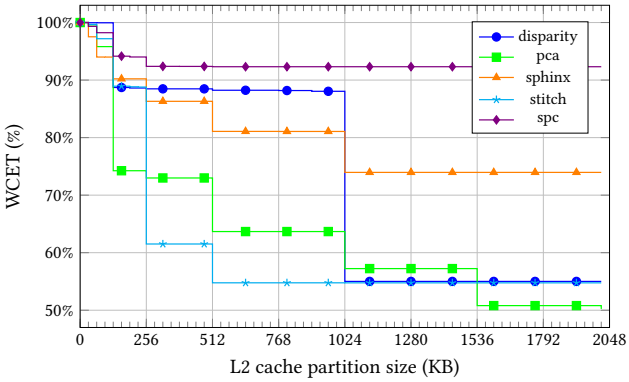


Figure 3: Execution time vs cache size for CortexSuite.

We consider a set of applications from the *CortexSuite* benchmark suite [54] for computer vision and machine learning. We assume a two-level set-associative cache hierarchy with *Least Recently Used* (*LRU*) eviction policy and line size of 64 B at both levels. Level L1 is split into L1d (for data) and L1i (for instructions), both with a fixed size of 32 KB and 4 ways. Level L2 of 16 ways is shared by instructions and data. We vary L2 cache size from 0 up to 2 MB, with steps of 32 KB. We ran 20 benchmarks and collected the number of cache misses (for L1 and L2), the number of data references, and the number of executed instructions. Then, assuming the characteristic of ARM Cortex A8 [30] (2 instructions per cycle, 1 cycle for L1 hit, 11 cycles for L1 miss, and 60 cycles for L2 miss) we got the execution times for each benchmark and for each cache partition size. Figure 3 shows a sample of the obtained profiles. Similar trends have been observed in [8] for *PARSEC* benchmark suite and in [31] for *SPEC2000* programs. The above simulation technique is similar to several works in real-time systems literature [40, 41, 45]. Nevertheless, the proposed cache usage optimization techniques can be used with any WCET analysis tool or measurement methodology.

Based on the benchmark profiles, we generate a large number of random task sets with different parameters: (i) task set size $n \in \{16, 32, 64\}$, (ii) available cache size $S \in \{1024, 2048, 4096\}$ KB, (iii) cache segment size $dS \in \{32, 128, 512\}$ KB, and (iv) task set base utilization $U \in [0.7, 1.6]$ with a step of 0.1. For each parameter combination, we randomly generate 20 task sets, which yields $3 \times 3 \times 3 \times 10 \times 20 = 5,400$ task sets in total. The generation of each task set follows three steps. First, we generate the periods T_i for n real-time tasks by uniformly sampling from $[10, 100]$ ms. Second, we generate the base utilization U_i of each task τ_i using *UUniFast* [9] such that the total base utilization of the task set equals the target base utilization (*i.e.*, $U = \sum_{i=1}^{n} U_i$). Finally, we randomly sample n benchmarks profiles and calculate each task's WCETs under different cache partitions according to the profile.

6.2 Experimental setup

In this paper, all the experiments are conducted on a workstation equipped with Intel Xeon Silver 4216 CPU running Linux. The proposed cache minimization algorithms are implemented in Python 3.10, and the IQCP is solved by a mathematical programming solver Gurobi 9.5.2 [29] with Python API. To complete the experiments within reasonable time, we set a run time limit of 3600 seconds. For the parameter L (*i.e.*, the maximum allowed number of schedulability checks) used in GLS and B&B, we set it to be proportional to nm and tried different values from $2nm$, $3nm$, and $4nm$. We observed that this parameter has a very low impact on both the cache usage and schedulability ratio, thus the lowest value $2nm$ is selected.

We run each comparison algorithm on each generated task set to evaluate its cache usage and schedulability. The cache usage of a test algorithm is recorded as the maximum cache available in the system, if the test set is not schedulable by the algorithm.

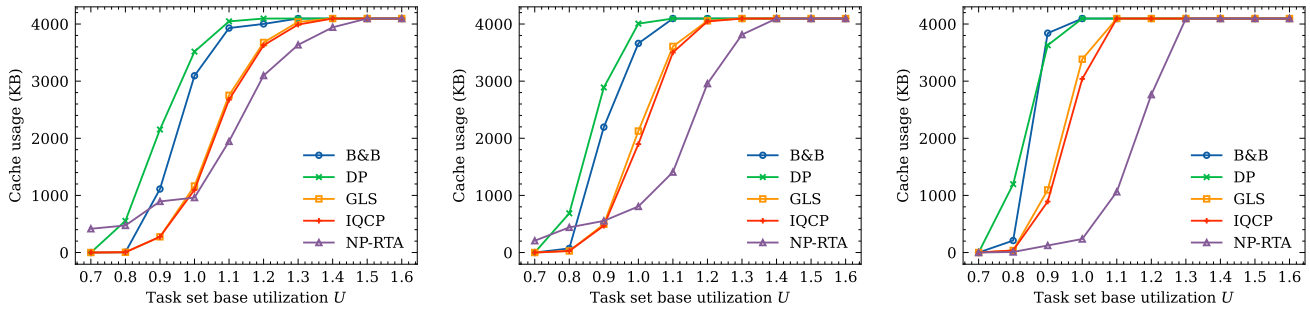
6.3 Experimental results

We present the results of our experimental study on cache usage, schedulability ratio, and run time. Specifically, in sections 6.3.1, 6.3.2, and 6.3.4, we compare the performance of various preemptive cache minimization approaches, including DP, B&B, GLS, and IQCP, and a non-preemptive approach combined with RTA (*i.e.*, NP-RTA). In Section 6.3.3, we compare the non-preemptive cache minimization approaches combined with different schedulability tests (*i.e.*, NP-RTA, NP-SINGLE, and NP-HYPER) and search techniques (*i.e.*, linear and binary search denoted by -L and -B, respectively).

6.3.1 *Cache usage for different U, n, and m.*

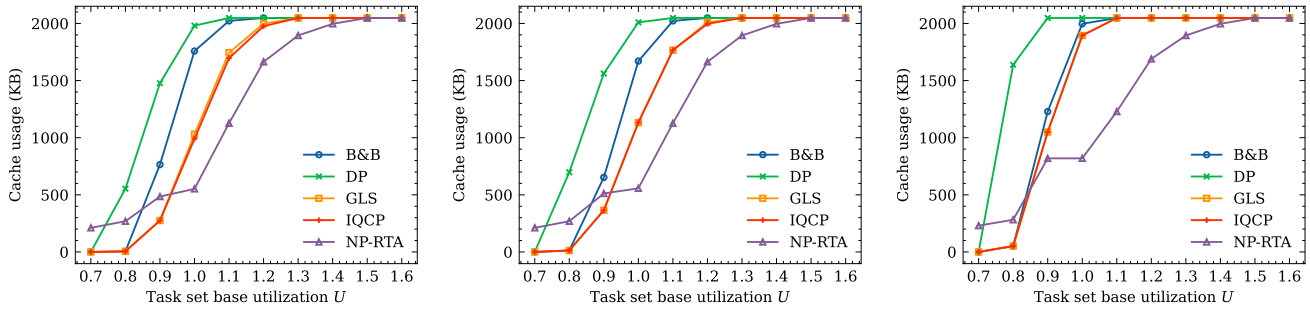
Figure 4a, 4b, and 4c present the cache usage with $n = 16$, 32, and 64, respectively, with $dS = 32$ and $S = 4096$ KB. The results support the following observations. First, the proposed GLS outperforms DP and B&B and achieves near-optimal cache usage compared to IQCP, which demonstrates the effectiveness of GLS. Second, the cache usage of the preemptive (*resp.*, non-preemptive) approaches increases (*resp.*, decreases) with the increase of n. Third, the non-preemptive approach NP-RTA uses more (*resp.*, less) cache than preemptive approaches for small (*resp.*, large) utilization.

The last two observations are explained as follows. First, non-preemptive tasks share the available cache as one single partition, leading to consistent WCET reduction regardless of the number of tasks. However, preemptive tasks need to divide the available cache into private partitions to avoid inter-task cache eviction, resulting in decreasing WCET reduction as n increases. Second, preemptive task sets are easily schedulable at low utilization, but non-preemptive scheduling can lead to unschedulability even at low utilization (the utilization bound drops to 0). As a result, the preemptive approach uses less cache when utilization is low, but non-preemptive is more efficient at higher utilization levels due to larger WCET reduction.



(a) $n = 16$. **(b)** $n = 32$. **(c)** $n = 64$.

Figure 4: Sensitivity of n with $dS = 32$ KB, $S = 4096$ KB.



(a) $dS = 32$ KB ($m = 64$). **(b)** $dS = 128$ KB ($m = 16$). **(c)** $dS = 512$ KB ($m = 4$).

Figure 5: Sensitivity of dS and m with $n = 16, S = 2048$ KB.

Figure 5a, 5b, and 5c present the cache usage with different dS, $n = 16$, and $S = 2048$ KB. The results of the experiment show that: (i) The cache usage of all algorithms increases with the increase of dS; (ii) The performance of B&B improves as dS increases. The reason for (i) is that dS represents the granularity of the cache segments, and as dS increases, precision in cache partitioning is lost. For (ii), it suggests that B&B can benefit more from smaller m compared with GLS. This is because B&B enumerates all possible cache allocation options without the knowledge guidance used in GLS. As a result, B&B can achieve a good performance for a coarse-grained cache partitioning but performs worse for a finer granularity.

6.3.2 *Schedulability ratio for different U and S.*

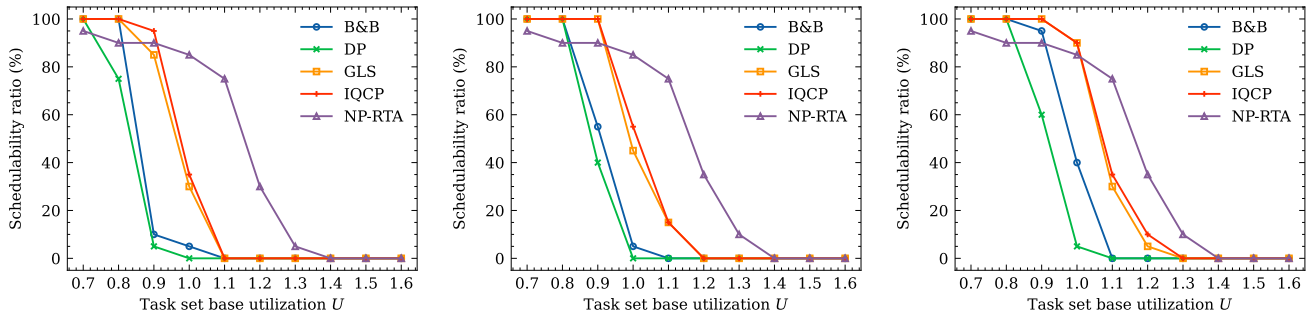
Figure 6a, 6b, and 6c present the schedulability ratio with different S, $n = 32$, and $dS = 32$ KB. The schedulability results show a similar trend as in cache usage: (i) GLS performs near-optimally compared to IQCP and outperforms other preemptive cache minimization methods; (ii) The non-preemptive method performs better than the preemptive ones for high utilization. Additionally, the schedulability ratio of the preemptive methods improves with increasing cache size, however, the non-preemptive method does not see further improvements beyond a cache size of 2048 KB. This is because non-preemptive tasks can share all available cache and our benchmarking experiments have shown that their WCETs do not improve with more than 2048 KB of cache.

6.3.3 *Impact of different non-preemptive schedulability tests.*

Figure 7a, 7b, and 7c respectively depict the cache usage, schedulability ratio and run time of the non-preemptive cache minimization approaches combined with different schedulability tests and search techniques. The results demonstrate that while NP-RTA has a longer run time (but still within the same order of magnitude), it is able to save cache usage by 24.9% and 41.4% in comparison to NP-SINGLE and NP-HYPER respectively. Furthermore, the linear search approaches run faster than the binary search for all schedulability tests under comparison.

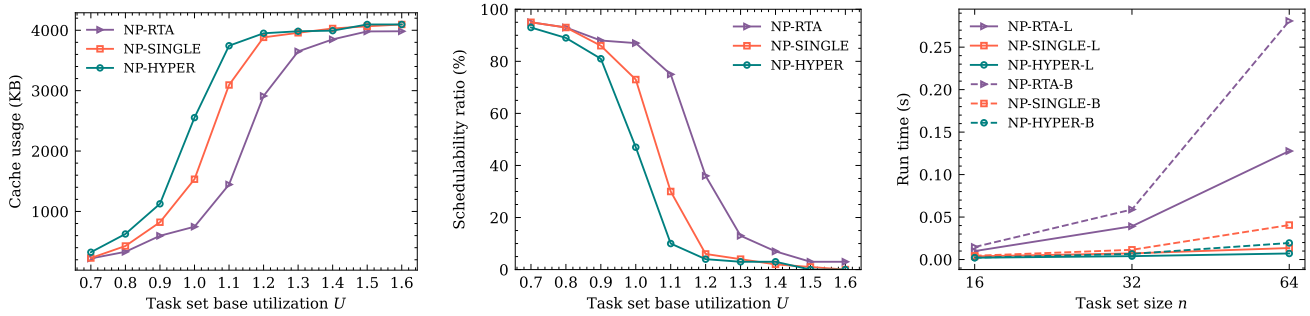
6.3.4 *Comparison of run time.*

The violin plot in Figure 8 illustrates the run time comparison of all proposed cache minimization algorithms. The plot displays the run time of solving 20 task sets, with each violin representing the run time distribution. The horizontal lines (from bottom to top) indicate the minimum, average, and maximum run time over the 20 runs. The results indicate that DP has the shortest run time among all the preemptive methods. B&B and GLS have similar run time as they both perform the same number of schedulability tests. IQCP runs faster than B&B and GLS when the number of tasks is less than or equal to 32, but its run time increases significantly and becomes 10 times larger than B&B and GLS when the number of



(a) $S = 1024$ **KB.** **(b)** $S = 2048$ **KB.** **(c)** $S = 4096$ **KB.**

Figure 6: Sensitivity of S with $n = 32$, $dS = 32$ KB.



(a) Cache usage. **(b) Schedulability ratio.** **(c) Run time.**

Figure 7: Comparison of non-preemptive approaches with different schedulability tests and search techniques.

tasks is 64. Furthermore, the non-preemptive approach is faster than the preemptive methods due to its lower computational complexity.

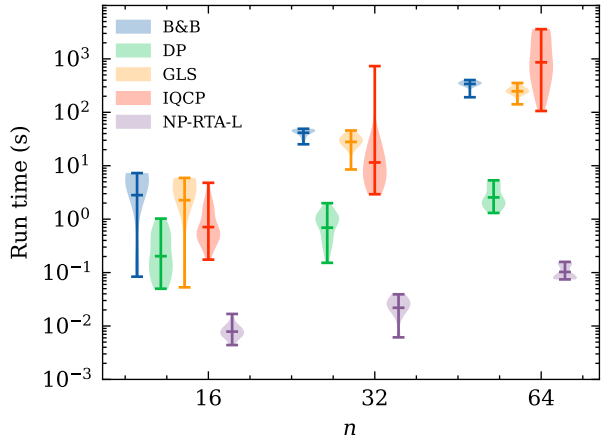


Figure 8: Comparison of run time with $dS = 32$, $S = 4096$ KB.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a comprehensive study of cache partitioning methods to minimize cache usage for real-time systems. We proposed efficient solutions for both preemptive and non-preemptive scheduling scenarios. For preemptive scheduling, we formulated the problem as an integer quadratically constrained program and proposed an efficient guided local search heuristic, a branch-and-bound search, and an efficient dynamic programming algorithm. For non-preemptive scheduling, we developed linear and binary searches coupled with different schedulability analyses. We evaluated the proposed methods using real-world benchmarks and found that our heuristic achieved an average optimality gap of 0.79%, with run time that was 0.1x that of a mathematical programming solver. Additionally, we demonstrated that the proposed heuristic outperforms existing cache partitioning techniques by reducing average cache usage by 39.15%. Furthermore, our results indicated that non-preemptive cache minimization methods can save more cache usage than preemptive methods for large task sets with high utilization.

In future work, we plan to investigate the potential benefits of combining non-preemptive and preemptive scheduling to further improve our cache usage minimization algorithms.

ACKNOWLEDGMENTS

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. Giovani Gracioli and Sérgio Arribas García were supported by Fundação de Desenvolvimento da Pesquisa - Fundep Rota 2030/Linha V 27192.02.01/2020.09-00.

REFERENCES

[1] Sebastian Altmeyer and Claire Burguière. 2009. A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 109–118.

[2] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. 2011. Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. In *IEEE Real-Time Systems Symposium (RTSS)*. 261–271.

[3] S. Altmeyer, R. I. Davis, and C. Maiza. 2012. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems* 48, 5 (2012), 499 – 526.

[4] Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I. Davis. 2016. On the Effectiveness of Cache Partitioning in Hard Real-Time Systems. *Real-Time Systems* 52, 5 (2016), 598–643.

[5] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. 1991. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings of 8th IEEE Workshop on Real-Time Operating Systems and Software*.

[6] Sanjoy Baruah and Pontus Ekberg. 2021. *An ILP Representation of Response-Time Analysis*. Technical Report. https://research.engineering.wustl.edu/~baruah/Submitted/2021-ILP-RTA.pdf

[7] Andrea Bastoni, Björn Brandenburg, and James Anderson. 2010. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of OSPERT*, Vol. 10. 33–44.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

[9] Enrico Bini and Giorgio C Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1 (2005), 129–154.

[10] Reinder J. Bril, Johan J. Lukkien, Rob I. Davis, and Alan Burns. 2006. Message response time analysis for ideal controller area network (CAN) refuted. In *International workshop on Real-Time Networks*.

[11] Georg Brüggen, Jian-Jia Chen, and Wen-Hung Huang. 2015. Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 90–101.

[12] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. 2008. 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)*. 101–110.

[13] Alan Burns and Sanjoy Baruah. 2008. Sustainability in Real-time Scheduling. *Journal of Computing Science and Engineering* 2, 1 (2008), 74–97.

[14] J.V. Busquets-Mataix, J.J. Serrano, R. Ors, P. Gil, and A. Wellings. 1996. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings Real-Time Technology and Applications*. 204–212.

[15] J.V. Busquets-Mataix, J.J. Serrano-Martin, R. Ors-Carot, P. Gil, and A. Wellings. 1996. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *Proceedings of Euromicro Workshop on Real-Time Systems*. 271–276.

[16] John M. Calandrino and James H. Anderson. 2008. Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 299–308.

[17] Sheng-Wei Cheng, Jian-Jia Chen, Jan Reineke, and Tei-Wei Kuo. 2017. Memory Bank Partitioning for Fixed-Priority Tasks in a Multi-core System. In *IEEE Real-Time Systems Symposium (RTSS)*. 209–219.

[18] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. 2007. Period Optimization for Hard Real-time Distributed Automotive Systems. In *ACM/IEEE Design Automation Conference (DAC)*. 278–283.

[19] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. 2007. Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised. *Real-Time Systems* 35, 3 (2007), 239–272.

[20] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. 2014. Predictable Flight Management System Implementation on a Multicore Processor. In *Embedded Real Time Software*.

[21] Rolf Ernst and Marco Di Natale. 2016. Mixed Criticality Systems—A History of Misconceptions? *IEEE Design & Test* 33, 5 (2016), 65–74.

[22] Farzad Farshchi, Prathap Kumar Valsan, Renato Mancuso, and Heechul Yun. 2018. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In *Euromicro Conference on Real-Time Systems (ECRTS)*, Vol. 106. 1:1–1:25.

[23] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. 2015. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *Comput. Surveys* 48, 2 (2015).

[24] Giovani Gracioli and Antônio Augusto Fröhlich. 2013. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 72–81.

[25] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. 2019. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 27:1–27:25.

[26] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-Aware Scheduling and Analysis for Multicores. In *Proceedings of the 7th ACM international conference on Embedded software*. 245–254.

[27] Zhishan Guo, Kecheng Yang, Fan Yao, and Amro Awad. 2020. Inter-Task Cache Interference Aware Partitioned Real-Time Scheduling. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC)*. 218–226.

[28] Zhishan Guo, Ying Zhang, Lingxiang Wang, and Zhenkai Zhang. 2017. Work-in-Progress: Cache-Aware Partitioned EDF Scheduling for Multi-core Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*. 384–386.

[29] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[30] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[31] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[32] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[33] Intel. 2015. *Improving real-time performance by utilizing cache allocation technology. White Paper.* Technical Report. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf

[34] M. Joseph and P. Pandya. 1986. Finding Response Times in a Real-Time System. *Comput. J.* 29, 5 (1986), 390–395.

[35] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. 2013. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 80–89.

[36] David Blair Kirk. 1989. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium (RTSS)*. 229–237.

[37] D. B. Kirk, J. K. Strosnider, and J. E. Sasinowski. 1991. Allocating SMART cache segments for schedulability. In *Proceedings of Euromicro Workshop on Real-Time Systems*. 41–50.

[38] Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. 2019. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–14.

[39] Angeliki Kritikakou, Claire Pagetti, Olivier Baldellon, Matthieu Roy, and Christine Rochange. 2014. Run-Time Control to Increase Task Parallelism In Mixed-Critical Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 119–128.

[40] Ohchul Kwon, Gero Schwäricke, Tomasz Kloda, Denis Hoornaert, Giovani Gracioli, and Marco Caccamo. 2021. Flexible Cache Partitioning for Multi-Mode Real-Time Systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1156–1161.

[41] Benjamin Lesage, David Griffin, Frank Soboczenski, Iain Bate, and Robert I. Davis. 2015. A Framework for the Evaluation of Measurement-Based Timing Analyses. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*. 35–44.

[42] Benjamin Lesage, Isabelle Puaut, and André Seznec. 2012. PRETI: Partitioned Real-Time Shared Cache for Mixed-Criticality Real-Time Systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems* (Pont à Mousson, France) *(RTNS '12)*. Association for Computing Machinery, New York, NY, USA, 171–180.

[43] ARM Limited. 2008. *PrimeCell Level 2 Cache Controller (PL310) Technical Reference Manual.* Technical Report. https://developer.arm.com/documentation/ddi0246/c/introduction/about-the-primecell-level-2-cache-controller--pl310-

[44] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20, 1 (1973), 46–61.

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

[46] Xing Pan and Frank Mueller. 2018. Controller-Aware Memory Coloring for Multicore Real-Time Systems. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC)*. 584–592.

[47] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Robert I. Davis, and Mateo Valero. 2011. IA3: An Interference Aware Allocation Algorithm for Multicore Hard Real-Time Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 280–290.

[48] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A Predictable Execution Model for COTS-Based Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 269–279.

[49] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. 2009. WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In *9th*

International Workshop on Worst-Case Execution Time Analysis, Vol. 10. 1–11.

[50] J.E. Sasinowski and J.K. Strosnider. 1993. A dynamic programming algorithm for cache memory partitioning for real-time systems. *IEEE Trans. Comput.* 42, 8 (1993), 997–1001.

[51] J. Seward, N. Nethercote, and J. Weidendorfer. 2008. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications.* Network Theory Ltd.

[52] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. 2008. Adaptive set pinning: managing shared caches in chip multiprocessors. *ACM Sigplan Notices* 43, 3 (2008), 135–144.

[53] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. 2013. Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses. In *IEEE 16th International Conference on Computational Science and Engineering*. 685–692.

[54] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. 2014. CortexSuite: A Synthetic Brain Benchmark Suite. In *International Symposium on Workload Characterization*.

[55] H. Tomiyama and N.D. Dutt. 2000. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign*. 67–71.

[56] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. 2013. Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 157–167.

[57] Jun Xiao, Sebastian Altmeyer, and Andy D. Pimentel. 2020. Schedulability Analysis of Global Scheduling for Multicore Systems With Shared Caches. *IEEE Trans. Comput.* 69, 10 (2020), 1487–1499.

[58] Jun Xiao, Yixian Shen, and Andy D. Pimentel. 2022. Cache Interference-Aware Task Partitioning for Non-Preemptive Real-Time Multi-Core Systems. *ACM Transactions on Embedded Computing Systems* 21, 3 (2022).

[59] Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, Yuhan Lin, Haoran Li, Chenyang Lu, and Insup Lee. 2019. Holistic Resource Allocation for Multicore Real-Time Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 345–356.

[60] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 381–392.

[61] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 155–166.

[62] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 55–64.

[63] Wei Zhang, Nan Guan, Lei Ju, Yue Tang, Weichen Liu, and Zhiping Jia. 2020. Scope-Aware Useful Cache Block Calculation for Cache-Related Pre-Emption Delay Analysis With Set-Associative Data Caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2333–2346.

[64] Ying Zhang, Zhishan Guo, Lingxiang Wang, Haoyi Xiong, and Zhenkai Zhang. 2017. Integrating Cache-Related Preemption Delay into GEDF Analysis for Multiprocessor Scheduling with On-chip Cache. In *IEEE Trustcom/BigDataSE/ICESS*. 815–822.