CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Trading Between Intra- and Inter-Task Cache Interference to Improve Schedulability

Syed Aftab Rashid
Geoffrey Nelissen
Eduardo Tovar

CISTER-TR-180803

2018/10/10

Trading Between Intra- and Inter-Task Cache Interference to Improve Schedulability

Syed Aftab Rashid, Geoffrey Nelissen, Eduardo Tovar

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: syara@isep.ipp.pt, grrpn@isep.ipp.pt, emt@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Caches help reduce the average execution time of tasks due to their fast operational speeds. However, caches may also severely degrade the timing predictability of the system due to intra- and inter-task cache interference. Intra-task cache interference occurs if thememory footprint of a task is larger than the allocated cache space or when two memory entries of that task are mapped to same space in cache. Inter-task cache interference occurs when memory entries of two or more distinct tasks use the same cache space. State-of-theart analysis focusing on bounding cache interference or reducing it by means of partitioning and by optimizing task layout in memory either focus on intra- or inter-task cache interference and do not exploit the fact that both intra- and inter-task cache interference can be interrelated.In this work, we show how one can model intra- and inter-task cache interference in a way that allows balancing their respective contribution to tasks worst-case response times. Since the placement of tasks in memory and their respective cache footprint determine the intra- and inter-task interference tasks may suï¬€er, we propose a technique based on cache coloring to improve taskset schedulability. Experimental evaluation performed using a set of benchmarks show that our approach result in up to 14% higher taskset schedulability than state-of-the-art approaches.

Trading Between Intra- and Inter-Task Cache Interference to Improve Schedulability

Syed Aftab Rashid
CISTER, ISEP, Polytechnic Institute of Porto, Portugal

Geoffrey Nelissen
CISTER, ISEP, Polytechnic Institute of Porto, Portugal

Eduardo Tovar
CISTER, ISEP, Polytechnic Institute of Porto, Portugal

ABSTRACT

Caches help reduce the average execution time of tasks due to their fast operational speeds. However, caches may also severely degrade the timing predictability of the system due to intra- and inter-task cache interference. Intra-task cache interference occurs if the memory footprint of a task is larger than the allocated cache space or when two memory entries of that task are mapped to the same space in cache. Inter-task cache interference occurs when memory entries of two or more distinct tasks use the same cache space. State-of-the-art analysis focusing on bounding cache interference or reducing it by means of partitioning and by optimizing task layout in memory either focus on intra- or inter-task cache interference and do not exploit the fact that both intra- and inter-task cache interference can be interrelated.

In this work, we show how one can model intra- and inter-task cache interference in a way that allows balancing their respective contribution to tasks worst-case response times. Since the placement of tasks in memory and their respective cache footprint determine the intra- and inter-task interference that tasks may suffer, we propose a technique based on cache coloring to improve task set schedulability. Experimental evaluations performed using Mälardalen benchmarks show that our approach results in up to 13% higher task set schedulability than state-of-the-art approaches.

1 INTRODUCTION

Caches bridge the performance gap between main memory and processor. Program data and instructions loaded in the cache are readily available to the processor and can be accessed in a few clock cycles. In comparison, when data or instructions are not in the cache and must be fetched from main memory, it results in a penalty of tens or even hundred of clock cycles [13]. While the use of caches can reduce the average execution time of tasks, it can also cause large execution variations depending on whether the instructions and data required by the tasks at run time are already present in the cache (cache hit) or not (cache miss). Moreover, as caches have a limited capacity, it is typical that not all data and instructions of a task (or a set of tasks) may simultaneously reside in the cache. This results in generating cache interference between different code segments of a same task and between sets of tasks sharing the same limited cache space. In the scientific literature, cache interference is broadly categorized into, (i) *intra-task cache interference*, that corresponds to main memory accesses when a task self evicts its own instructions/data from the cache, e.g., when instructions/data used by different code segments within a task are mapped to the same cache space; and (ii) *inter-task cache interference*, that corresponds to additional main memory reloads due to sharing of cache space between two or more distinct tasks, e.g., task τ_1 and task τ_2 may evict each others cache content if they use the same cache space. This evicted content may need to be reloaded again from the main memory resulting in extra main memory accesses.

To use caches in a predictable manner, many researchers have recognized and studied the problem of cache interference. Different approaches have been presented in literature to bound the intra- and inter-task cache interference and to integrate it into the schedulability analysis of a set of real-time tasks [1, 2, 8, 18, 24, 27, 29]. However, most of these works focus on either the intra- or the inter-task cache interference and do not exploit the fact that both intra- and inter-task cache interference can be interrelated. Moreover, a variety of approaches have also been presented in the literature to reduce cache interference by efficiently partitioning the cache among tasks [3, 7, 9, 15, 16, 30] or by optimizing the task layout in memory [10, 21]. However, these existing cache partitioning and task layout optimization approaches also mainly focus on reducing the inter-task cache interference and hence are not always beneficial in terms of task set schedulability. For example, the cache partitioning approaches are subjected to one basic problem: the available cache space may not be enough for each task to have its own independent (i.e., non-overlapping) cache partition. Also with cache partitioning, as the number of tasks increase, cache space that can be used for each individual task becomes always smaller. This reduced amount of cache space available to each task potentially increases its intra-task cache interference (i.e., the task may itself start to evict its own cache blocks) resulting in an increased execution time due to an increase in the number of main memory accesses. This may eventually cause the task to become unschedulable even though it does not suffer any inter-task cache interference. It has been identified [10, 21] that the approaches focusing on optimizing the task layout in memory may perform better in terms of schedulability in comparison to a full cache partitioning approach [3, 4]. The existing approaches to optimize task layout in memory [10, 21] changes task placements in memory to reduce the inter-task cache interference while allowing tasks an unconstrained use of the cache. However, we argue that even with an optimal layout of tasks in memory, allowing tasks an unconstrained use of cache may still result in higher inter-task cache interference, e.g., the cache block evictions of lower priority tasks caused by a higher priority task using the whole cache will be inevitable even with an optimal layout of tasks unless the cache space used by the higher priority task is reduced (i.e., potentially increasing the intra-task cache interference of the higher priority task to decrease the inter-task cache interference it may cause).

RTNS '18, October 10–12, 2018, Chasseneuil-du-Poitou, France

ACM ISBN 978-1-4503-6463-8/18/10…$15.00
https://doi.org/10.1145/3273905.3273924

In this work, we show how one can model intra- and inter-task cache interference in a way that allows balancing their respective contribution to tasks worst-case response times. We propose a technique optimizing the task layout in memory that result in improved task set schedulability. The main contributions of the paper are as follows: (1) We use a cache coloring[1] approach to optimize task layout in memory such that cache colors assigned to tasks are not strictly private but may be shared between tasks; (2) we model the impact of a given cache color assignment on different task parameters and show how intra- and inter-task cache interference can be upper-bounded when using cache coloring; (3) We present a *simulated annealing* algorithm to optimize the cache color assignment to tasks by re-allocating and re-sizing the cache colors assigned to tasks such that the task set's schedulability is achieved; and (4) we perform an experimental evaluation using a set of benchmarks showing that our approach results in up to 13% higher schedulability than state-of-the-art approaches.

2 SYSTEM MODEL AND NOTATIONS

We focus on single-core platforms with a single level of instruction cache[2]. The cache is assumed to be direct-mapped with k^{total} colors. Each color is uniquely numbered between 1 to k^{total}. The size of a cache color is denoted by k^{size} and is equal the number of successive sets in the cache that may be used by tasks assigned to that color. For simplicity, in this work we assume that the size of every cache color is the same. Note that this is a common practice in real systems [11].

We consider a fixed priority scheduling (e.g., Rate Monotonic or Deadline Monotonic) of a set of sporadic tasks. The task set τ comprises n tasks, i.e., $\tau = \{\tau_1, ..., \tau_n\}$. Each task τ_i is defined by a triplet $(C_i[k_i], T_i, D_i)$, where $C_i[k_i]$ is a vector of length k^{total} that contains the worst-case execution time of task τ_i in isolation assuming k_i contiguous cache colors are assigned to τ_i. Note that k_i represents the *number* of cache colors used by τ_i, whereas the *set* of cache colors assigned to τ_i is denoted by ck_i. The minimum inter-arrival time of τ_i is T_i and D_i is its relative deadline. We assume that the tasks have constrained deadlines, i.e., $D_i \leq T_i$. We further decompose each task's WCET in two independent terms bounding its processing and memory access demand, respectively. The worst-case processing demand PD_i denotes the worst-case execution time of τ_i considering that every memory access is a cache hit. Consequently, it only accounts for execution requirements of the task and does not include the time needed to fetch data and instructions from the main memory. $MD_i[k_i]$ is the worst-case memory access demand (in terms of time) of any job of task τ_i executing in isolation and assuming that k_i contiguous cache colors are assigned to τ_i. It is usually assumed that $C_i[k_i]$ is non-increasing with k_i, i.e., $k_i < k_i + 1 \implies C_i[k_i] \geq C_i[k_i + 1]$. However, we note that since PD_i is independent of the number of cache colors assigned to τ_i, it is the worst-case memory access demand $MD_i[k_i]$ which must be defined as a non-increasing function w.r.t. the number of cache colors assigned to τ_i, i.e., $k_i < k_i + 1 \implies MD_i[k_i] \geq MD_i[k_i + 1]$. Note that PD_i and $MD_i[k_i]$ may not necessarily be experienced on the same execution path of τ_i. Therefore, it holds that $C_i[k_i] \leq PD_i + MD_i[k_i]$. Furthermore, we assume that the values of $C_i[k_i]$, PD_i and $MD_i[k_i]$ can be calculated using a static timing analysis tool such as Heptane[3].

The worst-case response time (WCRT) of task τ_i, denoted by R_i, is defined as the longest time between the arrival and the completion of any job of τ_i. The worst-case reload time of a cache block from main memory is denoted by d_{mem}. For notational convenience, we use $\text{hp}(i)$ to denote the set of tasks with priorities higher than that of τ_i. Similarly, $\text{lp}(i)$ to denote the set of tasks with priorities lower than that of τ_i and $\text{hep}(i)$ denotes the set of tasks with priorities higher than or equal to that of τ_i (i.e. $hep(i)$ includes τ_i). Finally, $\text{aff}(i, j) = \text{hep}(i) \cap \text{lp}(j)$ denotes the set of intermediate tasks that can execute during the response time of τ_i but may also be preempted by a given higher priority task τ_j.

3 BACKGROUND

WCRT based schedulability analysis for fixed priority preemptive systems was first presented in [14] and is given as

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \tag{1}$$

Eq. (1) uses the worst-case execution time (WCET) of tasks in isolation to account for the interference task τ_i may suffer due to preemptions by all higher priority tasks in $\text{hp}(i)$. However, Eq. (1) did not explicitly consider cache interference except for the cache analysis performed during the WCET calculation of tasks. Eq. (1) has therefore been extended in several works (e.g., [1, 8, 25]) to account for the inter-task cache interference due to *Cache Related Preemption Delays (CRPDs)*. CRPDs are delays in execution time of a lower priority task τ_i due to preemptions by higher priority tasks in $\text{hp}(i)$, e.g., when a lower priority task τ_i is preempted by a higher priority task $\tau_j \in \text{hp}(i)$, the preempting task τ_j may evict cache blocks of the preempted task τ_i that has to be reloaded after task τ_i resumes its execution. These extra cache reloads during the execution of task τ_i are termed as CRPDs. CRPD of task τ_i due to preemption by a higher priority task $\tau_j \in \text{hp}(i)$ is usually denoted by $\gamma_{i,j}$. The WCRT analysis accounting for inter-task cache interference due to CRPDs is presented in [1] and is given by

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times (C_j + \gamma_{i,j}) \tag{2}$$

where $\gamma_{i,j}$ is usually calculated by categorizing the memory access patterns of tasks into *useful cache blocks* (UCBs) and *evicting cache blocks* (ECBs). These UCBs and/or ECBs are then used to upper-bound the contribution of $\gamma_{i,j}$ to the WCRT of τ_i. Lee et al. [18] first introduced the notion of useful cache blocks (UCBs) and defined it as *"a memory block m is called a useful cache block (UCB) at program point P , if it is cached at P and will be reused at program point Q that may be reached from P without eviction of m"*. Similarly, Busquets et al. [8] introduced the notion of evicting cache blocks (ECBs) and defined it as *"any cache block accessed during the execution of a task and which can then evict the memory block cached by another task is called an evicting cache block (ECB)"*.

A number of methods have been proposed in the literature [1, 2, 8, 18, 25, 27, 28] for computing $\gamma_{i,j}$ under fixed priority preemptive scheduling (FPPS) using the set of UCBs and/or ECBs. However, the ECB/UCB-union [1, 27] and the ECB/UCB multi-set approaches [2] dominate all the state-of-the-art approaches for CRPD calculation.

[1]Cache coloring works by controlling the mapping between the physical addresses referenced by tasks and their corresponding cache entries. Common bits between the physical page number and the cache set index are designated as a cache color index. This effectively divides the cache into different partitions based on their color index.

[2]Note that the cache level being considered here may not be the L1 but the L2 instead. We then consider that the intra- and inter-task interference in L1 is factored in the tasks' worst-case execution times.

[3]https://team.inria.fr/pacap/software/heptane/

In recent work, Rashid et al. [23, 24] proposed that CRPDs may not be enough to model the inter-task cache interference. They showed that it is not only the lower priority task τ_i that may suffer inter-task cache interference (i.e., CRPD) due to the execution of higher priority tasks $\in \text{hp}(i)$ but also the higher priority tasks $\tau_j \in \text{hp}(i)$ that may suffer inter-task cache interference in terms of *Cache Persistence Reload Overhead (CPRO)* due to the execution of tasks in $\text{hep}(i) \setminus \tau_j$. CPRO of a higher priority task $\tau_j \in \text{hp}(i)$ executing during the response time of a lower priority task τ_i is usually denoted by $\rho_{j,i}$ and is formally defined as [24] *"the maximum memory reload overhead suffered by a task τ_j due to evictions of its persistence cache blocks (PCBs) by tasks in* $\text{hep}(i) \setminus \tau_j$ *while τ_j is executing during the response time of τ_i"* where PCBs are defined as follows. *"A memory block of a task τ_j is persistent if once loaded by τ_j, it will never be invalidated or evicted from the cache when τ_j executes in isolation"*. The set of PCBs and ECBs of tasks is used to calculate CPRO $\rho_{j,i}$ under the CPRO-union or the multi-set approaches [24].

Rashid et al. [23, 24] also showed that thanks to PCBs subsequent jobs of a task may re-use most of the data and instructions that were already loaded in the cache during the execution of its previous jobs. Therefore, if all PCBs of a task τ_i were loaded in the cache by a previous job of τ_i, the memory demand of subsequent jobs of τ_i can be much lower than the worst-case memory demand of τ_i in isolation. This type of memory demand is called the residual memory demand of τ_i and is originally defined in [24] as *"the worst-case memory demand of any job of a task assuming all its PCBs are already loaded in the cache"*. As in this work, we propose a cache coloring approach where the residual memory demand of task τ_i also depends on the number of cache colors assigned to τ_i, i.e., k_i. Hence, in the remaining of the paper we will denote the residual memory demand of task τ_i by $MD_i^r[k_i]$. Effectively, the total memory demand $\hat{MD}_i(t)$ of a task τ_i within a time window of length t when τ_i executes in isolation is defined as [23, 24]

$$\hat{MD}_i(t) = \min\left\{\left\lceil \frac{t}{T_i}\right\rceil \times MD_i[k_i]\,;\left\lceil \frac{t}{T_i}\right\rceil \times MD_i^r[k_i] + |PCB_i| * d_{mem}\right\} \quad (3)$$

Furthermore, Rashid et al. [23, 24] also presented the WCRT analysis for FPPS that accounts for the inter-task cache interference due to both CRPD and CPRO and showed that their WCRT dominates the state-of-the-art WCRT analysis that only accounts for the inter-task cache interference due to CRPDs. The WCRT of a task τ_i is calculated in [23, 24] as

$$R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left(\left\lceil \frac{R_i}{T_j}\right\rceil \gamma_{i,j} + \min\left\{ \left\lceil \frac{R_i}{T_j}\right\rceil C_j\,; \left\lceil \frac{R_i}{T_j}\right\rceil PD_j + \hat{MD}_j(R_i) + \rho_{j,i} \right\} \right) \quad (4)$$

where $\gamma_{i,j}$ and $\rho_{j,i}$ bounds the CRPD and CPRO considering the pair of tasks τ_i and τ_j respectively. For more information on the formulation of Eq. (3) and (4), readers are referred to [24].

4 CACHE INTERFERENCE AWARE WCRT ANALYSIS

In this work, we calculate the WCRT of a task τ_i using a similar equation as presented in [23, 24] (i.e., Eq. (4)). However, we explicitly consider the intra- and inter-task cache interference suffered by tasks during the response time R_i of task τ_i, i.e.,

$$R_i = C_i^{min} + CI_i^{intra,k_i} + \sum_{\forall j \in \text{hp}(i)} \left(\min\left\{ \left\lceil \frac{R_i}{T_j}\right\rceil \left(C_i^{min} + CI_j^{intra,k_j}\right)\,; \left\lceil \frac{R_i}{T_j}\right\rceil PD_j + \hat{MD}_j(R_i) + CI_{j,i}^{inter,\rho}(R_i) \right\} + CI_{i,j}^{inter,\gamma}(R_i) \right) \quad (5)$$

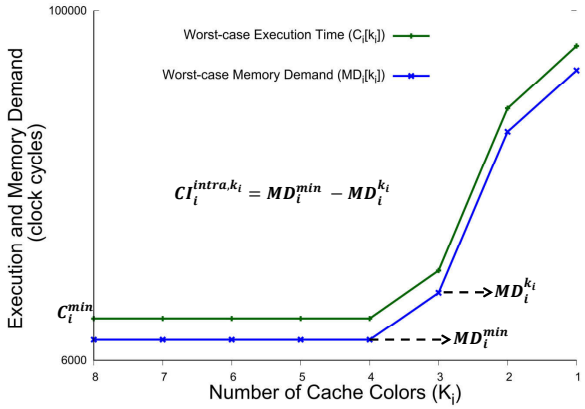


Figure 1: Increase in execution and memory demand of task τ_i due to reduction in number of cache colors assigned to τ_i.

In Eq. (5), C_i^{min} denotes the worst-case execution time of task τ_i in isolation assuming τ_i is allocated a cache of infinite size (or more practically, the total cache space assigned to task τ_i is greater or equal to the size of τ_i in main memory). The intra-task cache interference of τ_i w.r.t the number of cache colors k_i assigned to τ_i is denoted by CI_i^{intra,k_i} as intra-task interference impacts only the execution time of τ_i itself. Similarly, the intra-task cache interference CI_j^{intra,k_j} of each higher priority task $\tau_j \in hp(i)$ executing during the response time of τ_i is considered in the higher priority interference term within the sum on higher priority tasks. Moreover, $CI_{i,j}^{inter,\gamma}(R_i)$ denotes the inter-task cache interference in terms of CRPD that task τ_i may suffer during its response time due to preemptions by all higher priority tasks in $\text{hp}(i)$ and $CI_{j,i}^{inter,\rho}(R_i)$ bounds the inter-task cache interference in terms of CPRO that each higher priority task $\tau_j \in \text{hp}(i)$ may suffer during the response time of τ_i. Note that $\hat{MD}_j(R_i)$ in Eq. (5) is calculated in a similar manner to Eq. (4) (i.e., using Eq. (3)). Since $\hat{MD}_j(R_i)$ is a function of $MD_j[k_j]$, $MD_j^r[k_j]$ and the number of PCBs of τ_j, which are in turn functions of the number of cache colors k_j assigned to τ_j therefore, $\hat{MD}_j(R_i)$ directly considers the intra-task interference of all jobs of τ_j executing during the response time of τ_i.

In the following sections, we detail how the total intra- and inter-task cache interference can be bounded under the cache coloring approach presented in this paper.

5 INTRA-TASK CACHE INTERFERENCE

Intra-task cache interference represents contention between different code segments of a task that are mapped to the same cache space. If the cache space allocated to a task is not sufficient to hold all its instructions/data, a task may self-evict its own cache content resulting in higher main memory access demand even when the task is executing in isolation.

For a task τ_i its intra-task cache interference depends on the cache space or the number of cache colors k_i assigned to τ_i. Consider the plot of worst-case execution time ($C_i[k_i]$) and the worst-case memory demand ($MD_i[k_i]$) of task τ_i with respect to the number of cache colors k_i assigned to τ_i as shown in Fig. 1. The plot shows the actual variation in the worst-case execution time and the worst-case memory demand of the benchmark fdct of the Mälardalen benchmark suite [12], when the number of cache colors k_i assigned

to that task are varied in a descending order from 8 to 1. The values in Fig. 1 were obtained using Heptane for a cache with 8 cache colors, each having a size of 512 Bytes.

Fig. 1 shows that when the number of cache colors (or cache space) assigned to task τ_i is greater or equal to the size of τ_i in main memory (i.e., for $k_i \geq 4$), the worst-case execution time ($C_i[k_i]$) and the worst-case memory demand ($MD_i[k_i]$) of τ_i is minimum, i.e., $C_i[k_i] = C_i^{min}$ and $MD_i[k_i] = MD_i^{min}$ for $k_i \geq 4$, where MD_i^{min} represents the worst-case memory access demand of task τ_i in isolation assuming τ_i is allocated an infinite cache size. Effectively, for $k_i \geq 4$ τ_i will suffer no intra-task cache interference.

We can also observe from the plot in Fig. 1 that by decreasing the number of cache colors k_i assigned to τ_i, its worst-case execution time ($C_i[k_i]$) and the worst-case memory demand ($MD_i[k_i]$) tend to increase. This increase in $C_i[k_i]$ and $MD_i[k_i]$ is due to an increase in the intra-task cache interference of τ_i mainly because by reducing the number cache colors k_i, the number of UCBs of task τ_i may also decrease, i.e., by decreasing the number of cache colors k_i (or the cache space) assigned to τ_i , cache blocks of τ_i that were previously mapped to different cache sets and were reused more than once before eviction may now map to the same cache set. Consequently, loading one cache block will evict the other thus resulting in reducing the number of cache blocks of τ_i that can be reused, i.e., the number of UCBs. Effectively, this reduction of the number of UCBs results in increasing $MD_i[k_i]$ of τ_i for $k_i < 4$. Therefore, the intra-task cache interference of a task directly relates to it worst-case memory access demand in the following manner

$$CI_i^{intra,k_i} = MD_i[k_i] - MD_i^{min} \tag{6}$$

The resulting intra-task cache interference of τ_i for a given cache color assignment k_i, i.e., CI_i^{intra,k_i}, is accounted for in the WCRT of τ_i (i.e., Eq. (5)) by explicitly adding CI_i^{intra,k_i} to C_i^{min} which is the worst-case execution time of τ_i in isolation assuming τ_i is allocated an infinite cache. However, we note that because CI_i^{intra,k_i} depends on $MD_i[k_i]$ and since $MD_i[k_i]$ may not necessarily be experienced on the same execution path of τ_i for different cache color assignments ck_i, it holds that $C_i[k_i] \leq C_i^{min} + CI_i^{intra,k_i}$. Hence, Eq. (6) provides a safe upper-bound on intra-task cache interference even for multi-paths programs.

6 INTER-TASK CACHE INTERFERENCE

Under FPPS, the inter-task cache interference a task τ_i may suffer due to higher priority tasks in $\text{hp}(i)$ is mainly categorized into two types, i.e., the inter-task cache interference due to CRPDs and the inter-task cache interference due to CPROs.

The inter-task cache interference in terms of CRPD results from the eviction of UCBs of τ_i due to preemptions by a higher priority task τ_j in $\text{hp}(i)$ and is denoted by $CI_{i,j}^{inter,\gamma}$. Whereas, the inter-task cache interference in terms of CPRO results from the eviction of PCBs of the higher priority task $\tau_j \in \text{hp}(i)$ due to the executions of all other tasks in the system (while τ_j executes during the response time of τ_i) and is denoted by $CI_{j,i}^{inter,\rho}$. In the following subsections, we explain how $CI_{i,j}^{inter,\gamma}$ and $CI_{j,i}^{inter,\rho}$ can be bounded under the cache coloring approach proposed in this paper.

6.1 Inter-Task Cache Interference due to CRPDs

As discussed in Section 3, a number of methods have been proposed in the literature [1, 2, 8, 18, 25, 27, 28] for computing the CRPD cost $\gamma_{i,j}$ under FPPS using the set of UCBs and/or ECBs. However, in this work, we focus on a UCB-union-like approach [27] to calculate the CRPD cost due to sharing of cache colors between several tasks. The UCB-union approach [27] uses intersection between the ECBs of the preempting task τ_j and the UCBs of all tasks in $aff(i,j)$ possibly affected by the preemption caused by τ_j to calculate $\gamma_{i,j}$. Formally,

$$\gamma_{i,j} = d_{mem} \times \left| \left(\bigcup_{\forall s \in \text{aff}(i,j)} UCB_s \right) \cap ECB_j \right| \tag{7}$$

where, ECB_j and UCB_s are the sets of ECBs and UCBs of task τ_j and τ_s, respectively.

However, when cache colors are being assigned to tasks, Eq. (7) cannot be used as is. This is mainly because when coloring tasks, any variation in the cache color of any task may potentially change the set of UCBs and ECBs of all tasks in τ. Indeed, the actual mapping of tasks within a cache color may not be known as it is handled by the cache controller.[4] Consequently, the actual set of ECBs/UCBs of tasks may not be known as they depend on the actual cache sets used by the tasks. For example, consider two tasks τ_i and τ_s sharing the same cache color ck, where ck comprises 4 cache sets, numbered from 1 to 4. If both τ_i and τ_s have 2 UCBs under this cache assignment, these UCBs can be mapped to any of the four cache sets depending on how τ_i and τ_s are mapped within ck by the cache controller, i.e., $UCB_i = \{1, 2\}$ and $UCB_s = \{3, 4\}$ or any other combinations with or without overlapping between UCB_i and UCB_s. Since the actual set of UCBs of tasks might not be known, using different set of UCBs of tasks in Eq. (7) may lead to different pessimistic/optimistic value of γ.

In order to bound the CRPD $\gamma_{i,j}$ under our cache coloring approach, we first determine the cache colors that may be affected when τ_i is preempted by a higher priority task $\tau_j \in \text{hp}(i)$. Assuming that the cache color assignment of tasks has already been done, i.e., τ_i and τ_j are assigned a set of ck_i and ck_j cache colors respectively.

We know from the UCB-union approach (Eq. (7)), that when a task τ_i is preempted by a higher priority task τ_j, the set of UCBs of all tasks in $\text{aff}(i,j)$ can be evicted. Similarly, when a task τ_i using a set of ck_i cache colors is preempted by a higher priority task τ_j whose assigned a set of ck_j cache colors, the cache colors used by all tasks in $\text{aff}(i,j)$ may be evicted. Therefore, the maximum number of cache colors that may be affected due to a single preemption of τ_i by τ_j is bounded by $k_{i,j}$, where

$$k_{i,j} = \left| \left(\bigcup_{\forall s \in \text{aff}(i,j)} ck_s \right) \cap ck_j \right| \tag{8}$$

Here, $k_{i,j}$ gives the worst-case number of cache colors that may suffer evictions as a result of a single preemption of τ_i by τ_j. Therefore, the product $k_{i,j} \times k^{size}$ can be used to upper bound the number of cache sets that may be evicted due to a single preemption of τ_i by τ_j. However, this bound can obviously be very pessimistic, mainly because it does not consider the actual number of UCBs in

[4] Most cache controllers [19, 20, 26, 31] work at the granularity of a memory page and can be controlled to make sure memory pages of a task map to the specified cache color. However, when sharing cache colors among tasks, memory pages of different tasks may map to the same cache color so changing the mapping of one task may affect the others, making it difficult to predict the actual placement of tasks in cache.

those cache sets and hence the actual number of memory blocks that must be reloaded from main memory after eviction.

To tightly bound the CRPD cost, both the number of potentially evicted cache colors, i.e., $k_{i,j}$, and the number of ECBs/UCBs of tasks must be considered. We know that under cache coloring the actual set of ECBs/UCBs, i.e., their mapping in cache, may not be known as they depend on the actual cache sets assigned to tasks. However, their *number* only depends on the number of cache colors assigned to tasks rather than the actual cache sets assigned to those tasks. Therefore, let $UCB_i(k_i)$ and $ECB_i(k_i)$ be defined as

- $UCB_i(k_i)$: The maximum number of UCBs[5] of task τ_i when it is assigned k_i cache colors.
- $ECB_i(k_i)$: The maximum number of ECBs of task τ_i when it is assigned k_i cache colors.

Effectively, the CRPD cost due to a single preemption of τ_i by τ_j can be bounded using the notion of $UCB_i(k_i)$ and $ECB_i(k_i)$.

Lemma 1. The CRPD cost due to a single preemption of a lower priority task τ_i by a higher priority task τ_j is bounded by $\gamma^{col}_{i,j}$, i.e.,

$$\gamma^{col}_{i,j} = d_{mem} \times \min\left\{ \sum_{\forall s \in \text{aff}(i,j)} \left(UCB_s(k_s) \times V_{s,j}\right); ECB_j(k_{i,j}) \right\} \quad (9)$$

where $V_{s,j} = 1$ if $|ck_s \cap ck_j| > 0$ and $V_{s,j} = 0$, otherwise.

Proof. We prove that both $\sum_{\forall s \in \text{aff}(i,j)} \left(UCB_s(k_s) \times V_{s,j}\right)$ and $ECB_j(k_{i,j})$ are upper bounds on the CRPD cost $\gamma^{col}_{i,j}$. Therefore, the minimum between the two is also an upper bound on $\gamma^{col}_{i,j}$.

(1). From the UCB-union approach (Eq. (7)), it follows that when task τ_i is preempted by a higher priority task τ_j, the set of UCBs of all tasks in aff(i,j) may be evicted. However, when using cache coloring the actual set of UCBs of a task $\tau_s \in$ aff(i,j) may not be known. Instead, we know the maximum number of UCBs of τ_s, i.e., $UCB_s(k_s)$, for a given cache color assignment ck_s with size k_s. Also due to cache coloring, τ_j can only evict UCBs of a task $\tau_s \in$ aff(i,j) only when $|ck_s \cap ck_j| > 0$ (i.e., $V_{s,j} = 1$). Hence, the total number of UCBs among all tasks in aff(i,j) that can be evicted by τ_j is bounded by $\sum_{\forall s \in \text{aff}(i,j)} \left(UCB_s(k_s) \times V_{s,j}\right)$. Therefore, for a single preemption of τ_i by τ_j, $\sum_{\forall s \in \text{aff}(i,j)} \left(UCB_s(k_s) \times V_{s,j}\right)$ upper bounds the CRPD cost $\gamma^{col}_{i,j}$.

(2). The ECB-only approach [8, 28] implies that the number of ECBs of the preempting task upper bounds the total CRPD cost that a task may cause, i.e., for a single preemption of τ_i by τ_j the number of ECBs of τ_j also upper bounds the CRPD cost. However, due to cache coloring not all cache colors used by τ_j, i.e., k_j, may overlap with cache colors used by task τ_i (and by tasks in aff(i,j)) except for $k_{i,j}$ cache colors (i.e., Eq. (8)).

Hence, the maximum number of ECBs of τ_j in the $k_{i,j}$ overlapping cache colors used by tasks in aff(i,j), i.e., $ECB_j(k_{i,j})$, upper bounds the CRPD cost $\gamma^{col}_{i,j}$ from τ_j's perspective.

The lemma follows. □

For a single preemption of τ_i by τ_j, the CRPD cost can be bounded using Lemma 1. However as we will now prove, the actual time taken to reload all UCBs of τ_i from the main memory is also bounded by the change in the worst-case memory demand of task τ_i w.r.t the number of cache colors k_i assigned to τ_i.

[5]The maximum number of ECBs, UCBs and PCBs of a task for a given cache color assignment can be computed using any static timing analysis tool such as Heptane.

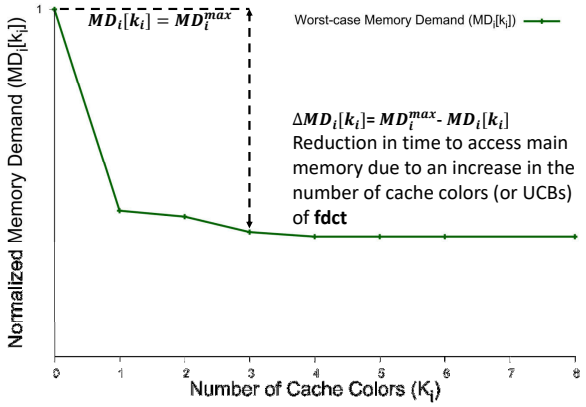


Figure 2: Worst-case memory demand $MD_i[k_i]$ of task τ_i w.r.t the number of cache colors assigned to τ_i.

To illustrate, let MD_i^{max} be the maximum worst-case memory demand of τ_i when there is no cache assigned to τ_i (i.e., $k_i = 0$). Now, consider the example plot of main memory access demand $MD_i[k_i]$ of a task τ_i shown in Fig. 2. The plot shows the normalized worst-case memory access demand of the fdct benchmark of the Mälardalen benchmark suite when the number of cache colors k_i assigned to that task varies. The values reported in Fig. 2 were obtained using the same cache configuration as in Fig. 1.

Fig. 2 shows that for $k_i = 0$ the worst-case memory access demand of τ_i is maximum, i.e., $MD_i[k_i] = MD_i^{max}$. Also, for $k_i = 0$ since no cache space is assigned to τ_i there cannot be any useful cache blocks, i.e., $UCB_i = \emptyset$. Moreover, since $MD_i[k_i]$ is a non-increasing function w.r.t. the number of cache colors k_i, we observe that by increasing k_i, $MD_i[k_i]$ is decreasing.

This decrease in $MD_i[k_i]$ of task τ_i is due to an increase in its number of UCBs, i.e., by increasing the number of cache colors k_i (or cache space) assigned to τ_i, more instructions/data of τ_i may remain cached and therefore reused without having to reload them from main memory. This effectively increases the number of UCBs of τ_i, leading to a reduction in its worst-case memory access demand. The change in the worst-case memory demand $MD_i[k_i]$ of τ_i due to an increase in the number of cache colors k_i assigned to τ_i can be bounded by $\Delta MD_i[k_i]$, where

$$\Delta MD_i[k_i] = MD_i^{max} - MD_i[k_i] \quad (10)$$

As the change in worst-case memory demand of τ_i is due to an increase in the number of accesses to UCBs of τ_i. Formally,

$$UCB_i(k_i) \times N_i \times d_{mem} \leq \Delta MD_i[k_i] \quad (11)$$

where N_i is the average number of times each UCB of τ_i is accessed while it is cached.

Since $\Delta MD_i[k_i]$ bounds the time to reload all UCBs of τ_i for a given cache color assignment k_i, it also bounds the total CRPD τ_i can suffer due to eviction of its UCBs by tasks in hp(i). However, we know from Lemma 1 that when task τ_i is preempted by a higher priority task $\tau_j \in$ hp(i), UCBs of all tasks in aff(i,j) can be evicted. Therefore, to bound the total CRPD τ_i may suffer due to preemptions by a task $\tau_j \in$ hp(i) the change in the worst-case memory demand of all tasks in aff(i,j) should be considered.

Lemma 2. The total CRPD cost suffered by a task τ_i due to preemptions by a higher priority task $\tau_j \in \text{hp}(i)$ is bounded by

$$\forall j \in \text{hp}(i) : \gamma_{i,j}^{tot} \leq \sum_{\forall s \in \text{aff}(i,j)} \Delta MD_s[k_s] \tag{12}$$

Proof. Assuming tasks are assigned priorities in ascending order such that task τ_{i-1} has a higher priority than τ_i, we prove by induction that Eq. (12) holds $\forall j \in \text{hp}(i)$.

Base Case: Consider τ_i and τ_{i-1} such that τ_{i-1} has a priority just above that of τ_i. Therefore, $\text{aff}(i, i-1) = \tau_i$.

The total CRPD that τ_i may suffer due to task τ_{i-1}, i.e., $\gamma_{i,i-1}^{tot}$, can never be larger than the time to reload all UCBs of τ_i N_i times from the main memory, i.e., the number of times UCBs of τ_i were accessed in cache when τ_i executes in isolation. Whereas, Eq. (11) implies that that time is bounded by $\Delta MD_i[k_i]$. Hence, for $j = i-1$, $\gamma_{i,j}^{tot} \leq \Delta MD_i[k_i]$.

Induction step: Consider another task τ_s having a priority higher than τ_i and assume that Eq. (12) holds for $j = s$, then Eq. (12) also holds for $j = s-1$.

For $j = s-1$, $\text{aff}(i, s-1) = \{\tau_i, ...\tau_s\}$, so using Eq. (9) we know that when τ_{s-1} preempts task τ_i it may evict UCBs of all tasks in $\text{aff}(i, s-1)$, i.e., $\{UCB_i(k_i), ...UCB_s(k_s)\}$. Also, by the same reasoning than above we know that the total CRPD every task in $\text{aff}(i, s-1) = \{\tau_i, ...\tau_s\}$ may suffer due to task τ_{s-1} is bounded by $\{\Delta MD_i[k_i], ...\Delta MD_s[k_s]\}$ respectively.

Therefore, it follows that for $j = s-1$, the total CRPD τ_i may suffer due to τ_j is bounded such that $\gamma_{i,j}^{tot} \leq \sum_{\forall s \in \text{aff}(i,j)} \Delta MD_s[k_s]$.

Therefore, by induction Eq. (12) holds for all $j \in \text{hp}(i)$. □

Since a higher priority task $\tau_j \in \text{hp}(i)$ can release $\left\lceil \frac{t}{T_j} \right\rceil$ jobs during a time window of length t and the CRPD caused by each of these jobs on τ_i can also be bounded using $\gamma_{i,j}^{col}$ (i.e., Eq. (9)), therefore the total CRPD τ_i may suffer due to τ_j, i.e., $\gamma_{i,j}^{tot}$, during a time window of length t is bounded such that $\gamma_{i,j}^{tot} \leq \left\lceil \frac{t}{T_j} \right\rceil \times \gamma_{i,j}^{col}$.

Consequently, The total inter-task cache interference in terms of CRPD suffered by τ_i due to a higher priority task $\tau_j \in \text{hp}(i)$ in a time interval of length t is upper bounded by $CI_{i,j}^{inter,\gamma}(t)$, where

$$CI_{i,j}^{inter,\gamma}(t) = \min\left(\left\lceil \frac{t}{T_j} \right\rceil \times \gamma_{i,j}^{col}; \sum_{\forall s \in \text{aff}(i,j)} \Delta MD_s[k_s]\right) \tag{13}$$

6.2 Inter-Task Cache Interference due to CPROs

Under FPPS, CPROs can be calculated using the CPRO-union or the multi-set approaches presented in [24]. However, in this work we will present a CPRO-union alike approach to bound CPRO under the proposed cache coloring approach. To calculate the CPRO $\rho_{j,i}$ of a task $\tau_j \in \text{hp}(i)$ executing during the response time of τ_i, the CPRO-union approach uses the set of PCBs of task τ_j and the set of ECBs of all tasks in $\text{hep}(i) \setminus \tau_j$, i.e.,

$$\rho_{j,i} = d_{mem} \times \left| PCB_j \cap \left(\bigcup_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} ECB_s \right) \right| \tag{14}$$

However, as already discussed for the CRPD calculation (Section 6.1), it is not possible to directly use the CPRO-union approach (i.e., Eq.(14)) under the task coloring configuration proposed in this paper, mainly because the actual set, i.e., their accurate placement in cache, of PCBs and ECBs may not be known.

Therefore, to bound the CPRO of a task τ_j (executing during the response time of τ_i) under our cache coloring approach, we use a similar technique to the one used in Section 6.1. We first bound the worst-case number of cache colors that may be evicted between two subsequent jobs of τ_j. Assuming τ_i and τ_j are assigned a set of ck_i and ck_j cache colors respectively, then the maximum number of cache colors of τ_j that can be evicted between its successive jobs due to the executions of all tasks in $\text{hep}(i) \setminus \tau_j$ during the response time of τ_i can be bounded by $k'_{j,i}$ calculated as follows.

$$k'_{j,i} = \left| ck_j \cap \left(\bigcup_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} ck_s \right) \right| \tag{15}$$

where $k'_{j,i}$ bounds the number of cache colors that can be affected by evictions between two successive jobs of τ_j. Therefore, the product $k'_{j,i} \times k^{size}$ bounds the maximum number of cache sets that can be evicted between two successive jobs of τ_j.

However, that is obviously pessimistic and to have a tighter bound on the CPRO in terms of the number of PCBs of τ_j that may be evicted between its two successive jobs, we define $PCB_i(k_i)$, i.e.,

- $PCB_i(k_i)$: The maximum number of PCBs of task τ_i when it is assigned k_i cache colors.

$PCB_i(k_i)$ can also be computed in a similar manner to $ECB_i(k_i)$ and $UCB_i(k_i)$ as detailed in Section 6.1. Furthermore, $PCB_j(k_j)$ and $ECB_i(k_i)$ can be used to bound the CPRO of a task $\tau_j \in \text{hp}(i)$ executing during the response time of τ_i using the following lemma

Lemma 3. $\rho_{j,i}^{col}$ bounds the CPRO or the maximum number of PCBs of task τ_j that may be evicted between two successive jobs of τ_j due to eviction of $k'_{j,i}$ cache colors by tasks in $\text{hep}(i) \setminus \tau_j$, where

$$\rho_{j,i}^{col} = d_{mem} \times \min\left\{ PCB_j(k_j); \sum_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} \left(ECB_s(k'_{j,i}) \times V_{s,j} \right) \right\} \tag{16}$$

where $V_{s,j} = 1$ if $|ck_s \cap ck_j| > 0$ and $V_{s,j} = 0$, otherwise.

Proof. We prove that both $PCB_j(k_j)$ and $\sum_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} (ECB_s(k'_{j,i}) \times V_{s,j})$ are upper bounds on the CPRO cost $\rho_{j,i}^{col}$. Therefore, a min between the two is also an upper bound on $\rho_{j,i}^{col}$.

(1). By definition of PCBs, the CPRO of a task is upper bounded by its number of PCBs. Hence, assuming τ_j is assigned k_j cache colors, the maximum number of PCBs of τ_j are given by $PCB_j(k_j)$. Therefore, the maximum CPRO one job of τ_j can suffer during the response time of τ_i is upper bounded by $PCB_j(k_j)$.

(2). The worst-case memory interference of any task $\tau_s \in \text{hep}(i) \setminus \tau_j$ on τ_j is when it loads all its ECBs between two subsequent jobs of τ_j. With cache coloring, if k_s cache colors are assigned to a task $\tau_s \in \text{hep}(i) \setminus \tau_j$, the maximum number of ECBs of τ_s that can be loaded between two jobs of τ_j are bounded by $ECB_s(k_s)$.

(3). However, τ_s can only evict PCBs of task τ_j only if $|ck_s \cap ck_j| > 0$ (i.e., $V_{s,j} = 1$) and not all cache colors used by τ_s (i.e., ck_s) may overlap with cache color used by τ_j except for $k'_{j,i}$ cache colors (see Eq. (15)). Effectively, $ECB_s(k'_{j,i}) \times V_{s,j}$ bounds the number of ECBs of τ_s that may overlap and potentially evict PCBs of τ_j.

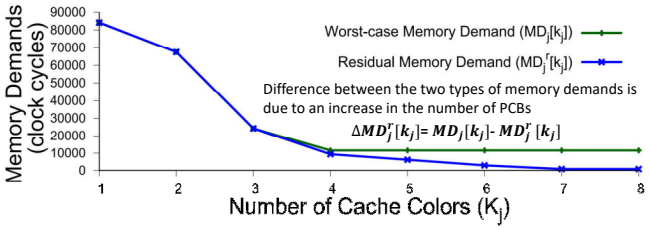


Figure 3: Variation in the worst-case and residual Memory demand of task τ_j w.r.t the number of cache colors assigned.

Since all $\tau_s \in \text{hep}(i) \setminus \tau_j$ may execute between two successive jobs of τ_j potentially evicting its PCBs. The worst-case memory interference of all $\tau_s \in \text{hep}(i) \setminus \tau_j$ on PCBs of τ_j is bounded by $\sum_{\forall \tau_s \in \text{hep}(i) \setminus \tau_j} ECB_s(\hat{k}_{j,i} \times V_{s,j})$. So the lemma follows. □

The CPRO (i.e., $\rho^{col}_{j,i}$) suffered by a single job of a higher priority task $\tau_j \in \text{hep}(i)$ executing during the response time of τ_i can be bounded using Lemma 3. However, since the CPRO accounts for the extra memory accesses of a task τ_j due to eviction of its PCBs, it may also depend on the memory demand of τ_j given that τ_j is assigned k_j cache colors.

To further illustrate this point, consider the example plot (i.e., Fig. 3) of a task τ_j representing the same (i.e., fdct) benchmark from the Mälardalen benchmark suite with the same cache configuration as used in Fig. 2. The plot in Fig. 3 shows two types of memory demands of task τ_j with respect to the number of cache colors (i.e., k_j) assigned to τ_j, i.e., the worst-case memory demand ($MD_j[k_j]$) and the residual memory demand ($MD^r_j[k_j]$).

From Fig. 3, we observe that when the number of cache colors assigned to τ_j are less than or equal to 3 , the worst-case memory demand $MD_j[k_j]$ of τ_j is equal to its residual memory demand $MD^r_j[k_j]$, showing that for $k_j \leq 3$, τ_j has no PCBs. However, by further increasing the cache colors assigned to τ_j (i.e., for $k_j > 3$), we can see an increasing difference between the worst-case memory demand $MD_j[k_j]$ and the residual memory demand $MD^r_j[k_j]$ of τ_j. This difference is due to an increase in the number of PCBs of τ_j and is denoted by $\Delta MD^r_j[k_j]$, where

$$\Delta MD^r_j[k_j] = MD_j[k_j] - MD^r_j[k_j] \tag{17}$$

$\Delta MD^r_j[k_j]$ corresponds to the reduction in time to access main memory due to an increase in the number of PCBs of τ_j. Therefore, $\Delta MD^r_j[k_j]$ effectively bounds the number of PCBs of τ_j given that τ_j is assigned k_j cache colors, i.e.,

$$PCB_j(k_j) \times N'_j \times d_{mem} \leq \Delta MD^r_j[k_j] \tag{18}$$

where N'_j is the average number of times each PCB of τ_j is accessed. Since $\Delta MD^r_j[k_j]$ bounds the number of PCBs of task τ_j, it also bounds the CPRO suffered by τ_j when it executes during the response time of a lower priority task τ_i.

Lemma 4. The CPRO due to the eviction of PCBs of a job of task $\tau_j \in \text{hp}(i)$ executing during the response time of a task τ_i, i.e., $\rho^{one}_{j,i}$, is upper bounded by the difference between the worst-case and the residual memory demand of τ_j, i.e.,

$$\rho^{one}_{j,i} \leq \Delta MD^r_j[k_j] \tag{19}$$

Proof.
(1). Eq. (16) implies that the CPRO of one job of task $\tau_j \in \text{hp}(i)$ executing during the response time of a task τ_i is upper bounded by the time to reload all PCBs of τ_j from main memory given a cache color assignment k_j, i.e., $\rho^{one}_{j,i} \leq PCB_j(k_j) \times d_{mem}$.
(2). Also, from Eq. (18) it follows that the time to reload all PCBs of τ_j for a given cache color assignment k_j is bounded by the difference between the worst-case and the residual memory demand of τ_j, i.e., $PCB_j(k_j) \times N'_j \times d_{mem} \leq \Delta MD^r_j[k_j]$.
(3). By definition of PCBs, $N'_j \geq 1$. So the lemma follows. □

Lemma 4 can be used to bound the CPRO of one job of task $\tau_j \in \text{hep}(i)$ executing during the response time of a task τ_i. However, we know that task τ_j may execute several times during the execution of τ_i therefore, the total inter-task cache interference in terms of CPRO suffered by τ_j while executing during the response time of τ_i can be bounded using the following theorem

Theorem 1. The total inter-task cache interference in terms of CPRO suffered by a higher priority task $\tau_j \in \text{hp}(i)$ due to evictions of its PCBs by tasks in $\text{hep}(i) \setminus \tau_j$ in a time interval of length t is bounded by $CI^{inter,\rho}_{j,i}$, where

$$CI^{inter,\rho}_{j,i}(t) = \left(\left\lceil \frac{t}{T_j} \right\rceil - 1\right) \times \min\left(\rho^{col}_{j,i}; \Delta MD^r_j[k_j]\right) \tag{20}$$

Proof.
(1). It is proved in [23] that in a time interval of length t at most $\left(\left\lceil \frac{t}{T_j} \right\rceil - 1\right)$ jobs of task τ_j can suffer CPRO.
(2). It implies that both $\rho^{col}_{j,i}$ (Eq. (16)) and $\Delta MD^r_j[k_j]$(by Lemma 4 and Eq. (17)) upper bound the CPRO suffered by one job of τ_j executing during the response time of τ_i. Therefore, the minimum between the two bounds is also an upper bound on the CPRO suffered by a single job of τ_j during the response time of τ_i.
The theorem directly follows from the two points above. □

7 OPTIMIZING CACHE COLOR ASSIGNMENT

In this section, we detail how we optimize the cache color assignment of tasks to balance the intra- and inter-task cache interference such that it results in improving task set schedulability. We have used a Simulated Annealing (SA) approach to optimize cache color assignment of tasks. Simulated annealing [17] is a meta-heuristic that allows to find a near optimal solution to an optimization problem in a reasonable computational time. Our SA-based cache coloring approach is given by Algorithm 1 (see Appendix A).

When allocating cache colors to tasks, the algorithm starts by assigning sequential cache colors to all n tasks in a given task set τ. Cache colors are assigned to tasks in priority order with the highest priority task first. Once the sequential cache color assignment is done, the algorithm checks the schedulability of each task in τ. If all tasks in task set τ are schedulable with the sequential cache color allocation (i.e., τ is schedulable), no changes are made to the cache color assignment of tasks and the algorithm returns true and exit. However, if τ was not schedulable with the sequential cache color allocation, cache color assignment of tasks is optimized using SA. The SA algorithm uses the sequential cache color assignment of tasks as the initial solution and then iteratively tries to improve it by randomly performing one of the following operations:

- **Re-allocate():** Swap the set of cache colors assigned to two distinct tasks. Namely two operations can be performed, (1) *swap-neighbors()*: swapping the set of cache colors assigned to two

neighboring tasks. This swap is based on the order of tasks in the main memory rather than their priorities ;(2) *swap-random()*: swap the set of cache colors of two randomly chosen tasks. These tasks may or may not be adjacent in main memory. If the chosen tasks are not adjacent in memory, cache color assignment of tasks in between them is also updated.

- **Shift-layout():** Increasing/decreasing the starting offset of a randomly chosen task in the main memory (i.e., shifting tasks right or left). To avoid creating gaps between tasks in main memory we essentially left/right shift all tasks in the main memory.
- **Re-size():** Randomly choose a task and re-allocate the number of cache colors assigned to that task, i.e., either by increasing or decreasing the number of cache colors assigned to that task.
  As we later show in Section 7.1, that re-sizing the cache space assigned to tasks can be very beneficial especially when the tasks have large cache footprints. Also, increasing/decreasing the number of cache colors assigned to tasks effectively allows to trade between the intra- and inter-task cache interference which may result in improving task set schedulability.

To evaluate different cache color assignments, the WCRT analysis (i.e., Eq. (5)) can be used at every iteration of the SA algorithm, i.e., checking the schedulability of all tasks in τ after performing any of the above mentioned operations. However, this may be computationally expensive. Also, the boolean result given by Eq. (5) can only distinguish between schedulable/unschedulable cache color assignments and does not provide any information about the impact of different cache color assignments on the intra- and inter-task cache interference suffered by the tasks. Therefore, to better quantify the quality of a cache color assignment of tasks and to guide the SA algorithm towards an optimal solution, we use the notion of *slack*. Slack S of a task τ_i is denoted by S_i and is defined as *"the difference between the relative deadline and the WCRT of τ_i"*, i.e., $S_i = D_i - R_i$, where R_i is calculated by considering the worst-case interference on τ_i by all higher priority tasks in hp(i), i.e., by setting $R_i == D_i$ in Equation (5). The total slack S^{tot} of task set τ is given as

$$S^{tot} = \sum_{i=1}^{n} w_i \times S_i \tag{21}$$

where w_i is the weight assigned to every $\tau_i \in \tau$ such that,

$$w_i = 0 \quad \text{if} \quad S_i \geq 0 \quad \text{and} \quad w_i = 1, \quad \text{otherwise.}$$

Note that only the tasks with a negative slack will be assigned a non-zero weight, i.e., $w_i = 1$. This is mainly because these are the tasks that were not schedulable for a given cache color assignment but may become schedulable by changing their cache color assignment. The total task set slack is calculated after randomly performing any of the above mentioned moves during every iteration of the SA algorithm. If the change in the total task set slack from the last iteration is positive then the new cache color assignment of tasks will always be accepted. However, even if the change in task set slack is negative the new cache color assignment of tasks may still be accepted depending on how negative the change is and the current temperature of the SA algorithm, i.e., if a randomly chosen probability between 0 and 1 is less than the probability of accepting the negative change, i.e., $e^{\frac{-\Delta Slack}{CurrentTemp}}$ (see Appendix A), then the new cache color assignment for τ will be accepted. Otherwise, the new cache color assignment will be discarded. After every iteration, the temperature of SA is reduced by multiplying it with a cooling factor until it reaches the desired temperature. The initial temperature, desired temperature and the cooling factor defines the maximum number of iterations for the SA algorithm. In general, when the temperature is high, the SA algorithm is more open to negative changes that may be useful to escape local minima.

7.1 Working Example

To evaluate the effectiveness of the SA-based cache color assignment approach detailed in the previous section, we performed a small experiment using a single task set comprised of 10 tasks from the Mälardalen benchmark suite [12] shown in Table 1, i.e., $\tau_1 = minmax$ to $\tau_{10} = bsort100$, where τ_1 has the highest priority. The selection of tasks was purely random and although these tasks may not represent a real task set, they do represent typical code found in real-time systems. For each task, the WCET $C_i[k_i]$, worst-case memory demand $MD_i[k_i]$, worst-case processing demand PD_i and the number of ECBs (i.e., $ECB(k_i)$), UCBs (i.e., $UCB(k_i)$) and PCBs (i.e., $PCB(k_i)$) were extracted using the Heptane static WCET analysis tool as presented in [23, 24]. Note that the values for $C_i[k_i]$, $MD_i[k_i]$ and PD_i in Table 1 are in clock cycles. The number of cache colors used by each task, i.e., k_i, were set such that $k_i = \left\lceil \frac{ECB_i(k_i)}{k^{size}} \right\rceil$. The target architecture was MIPS R2000/R3000 assuming an instruction cache with line size of 32 Bytes and the total cache size of 16kB such that the cache has a total of 32 cache colors, i.e., $k^{total} = 32$ with each color having a size of 512 Bytes, i.e., $k^{size} = 512$ Bytes. The block reload time d_{mem} was set to 10μs.

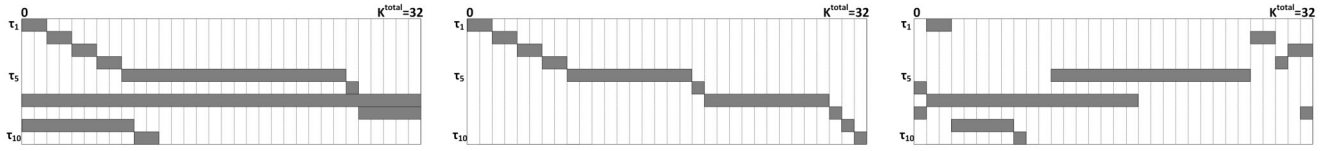
Table 1: task set parameters used in the working example

Name	$C_i[k_i]$	PD_i	$MD_i[k_i]$	k_i	T_i
minmax	2522	122	2400	2	14315
lcdnum	3440	984	2740	2	73143
cnt	10090	7191	3818	2	85816
ns	30149	28149	6172	2	169744
statemate	43344	10586	35257	18	636613
insertsort	7574	5974	2343	1	734873
nsichneu	316409	22009	294400	32	1889824
qurt	26141	9241	17713	5	2899034
fft	157880	123681	45816	9	6550339
bsort100	712289	710289	90893	2	267271122

The task set was created by fixing the core utilization at 0.8, with task utilizations generated using the UUnifast algorithm [6]. Task periods were set such that $T_i = C_i[k_i]/U_i$. All tasks had implicit deadlines with priorities assigned in deadline monotonic order. We checked task set schedulability using the following approaches:

- **No preemption cost:** The WCRT analysis was performed assuming there is no preemption cost (i.e., Eq. (1)).
- **SA-based cache color assignment:** Cache color assignment of tasks was optimized using the SA algorithm detailed in Section 7.
- **SA-based cache color assignment without re-sizing:** The SA algorithm was used to optimize cache color assignment of tasks however, re-size() operation was not permitted.
- **Sequential cache color assignment:** Tasks were assigned cache colors in a sequential manner with the highest priority task first.
- **Full cache partitioning:** The cache partitioning algorithm presented in [3] was used to assign independent non-overlapping cache colors (i.e., partitions) to all tasks.
- **SA-based cache color assignment without cache persistence** The SA algorithm was used to optimize cache color assignment of tasks without considering cache persistence.

We observed that the task set was schedulable only with two approaches, i.e., no preemption cost and the SA algorithm with re-sizing. All other approaches were not able to schedule the task



(a) Sequential cache color assignment. (b) Full cache partitioning. (c) SA-based cache color assignment with re-sizing.

Figure 4: Different cache color assignments of task set in Table 1.

set. The final cache color allocations for the sequential cache color assignment, full cache partitioning and the SA algorithm with re-sizing, are shown in Fig. 4a, 4b and 4c respectively.

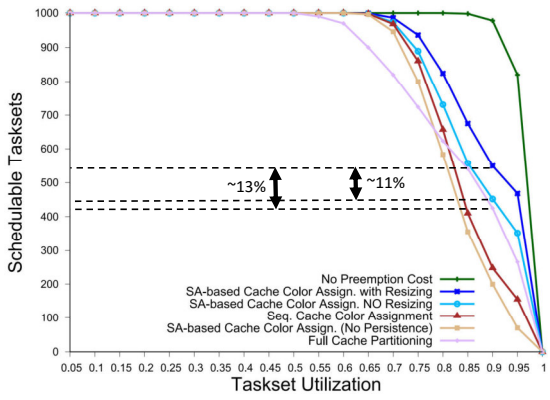
The sequential cache color assignment of tasks (see Fig 4a) was subjected to high inter-task cache interference (i.e., CRPD and CPRO), mainly because most cache colors were shared among tasks. This results in making the task set unschedulable. On the contrary, with full cache partitioning (see Fig 4b) there is no inter-task cache interference. However, the task set was still not schedulable due to an increase in the intra-task cache interference of some tasks that were assigned fewer cache colors than the actual number of cache colors needed by those tasks. The layout of tasks in cache resulting from the SA algorithm with re-sizing is shown in Fig 4c. The task set was schedulable mainly because the overall cache interference between tasks was reduced by trading between intra- and inter-task cache interference, e.g., the inter-task cache interference caused by τ_7 on all lower priority tasks (i.e., τ_8, τ_9 and τ_{10}) was reduced by increasing the intra-task cache interference of τ_7 (i.e., by reducing the number of cache colors used by τ_7). Note that the task set was also not schedulable using the SA algorithm without re-sizing. This shows that even with an optimized task layout, allowing tasks an unconstrained use of the cache may still result in higher inter-task cache interference that can make the task set unschedulable.

8 EXPERIMENTAL EVALUATION

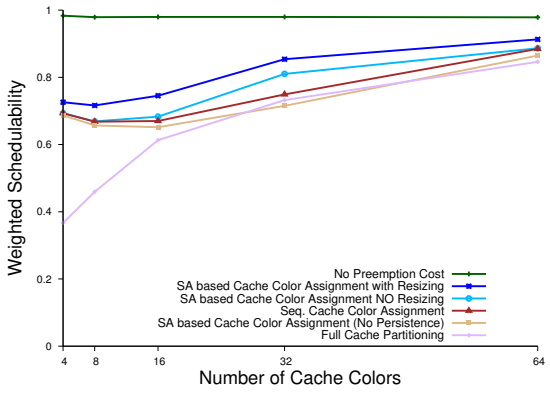
In this section, we evaluate how our proposed SA-based cache coloring approach performs in terms of schedulability in comparison to the state-of-the-art techniques. Experiments were performed using the Mälardalen benchmark suite with parameters $C_i[k_i]$, PD_i, $MD_i[k_i]$, $MD_i^r[k_i]$, $UCB_i(k_i)$, $ECB_i(k_i)$ and $PCB_i(k_i)$ extracted using Heptane for the same cache configuration as used in Section 7.1. The number of cache colors used by each task, i.e., k_i, were set such that $k_i = \left\lceil \frac{ECB_i(k_i)}{k^{size}} \right\rceil$. Each task was randomly assigned the values $C_i[k_i]$, PD_i, $MD_i[k_i]$, $MD_i^r[k_i]$, $UCB_i(k_i)$, $ECB_i(k_i)$, $PCB_i(k_i)$ and k_i of one of the analyzed benchmarks. Other task set parameters were randomly generated as follows. The default number of tasks was 10 with task utilizations generated using UUnifast [6]. Task periods were set such that $T_i = C_i[k_i]/U_i$. Task deadlines were implicit and priorities were assigned in deadline monotonic order.

We conducted different experiments by varying core utilization, number of cache colors and number of tasks. Schedulability analysis was performed using the same task sets for all the approaches detailed in Section 7.1 using their respective WCRT analysis.

1) Core Utilization: In this experiment, we randomly generated 1000 task set (each comprised of 10 tasks) at different core utilizations varied from 0.05 to 1 in steps of 0.05. Fig. 5a shows an average number of task sets that were schedulable using all the analyzed approaches against the total core utilization. The green line marked



(a) Varying core utilizations



(b) Varying the number of cache colors (or cache size)

Figure 5: Schedulability w.r.t core utilization and cache size

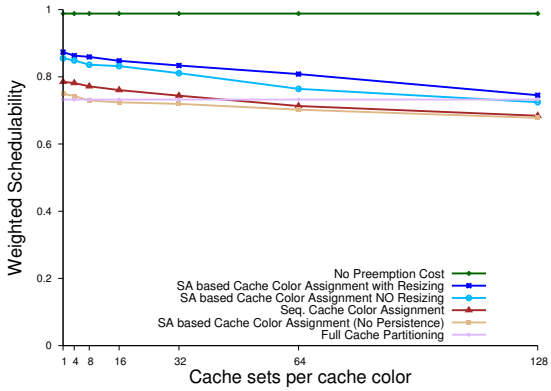
as "No preemption cost" is an upper bound on the maximum number of task sets that were schedulable without considering any CRPD/CPRO. Fig. 5a shows that the proposed SA-based cache color assignment with/without re-sizing was able to schedule more task sets than all the other approaches. Also, we note that while initially the full cache partitioning approach performs worst however, at higher core utilizations it tends to outperform the sequential cache color assignment and the SA-based cache color assignment (no persistence) approach. This is mainly because at higher core utilizations, task periods become smaller resulting in higher inter-task cache interference. It is due to higher inter-task cache interference that at core utilizations of 0.85 and 0.9 the difference between the full cache partitioning approach and the SA-based cache color assignment without re-sizing is minimal. However, the SA-based cache color assignment with re-sizing counters this increase in

inter-task cache interference by trading intra-task cache interference effectively resulting in much higher schedulability even at higher core utilizations. For example at a utilization of 0.9, the SA-based cache color assignment with re-sizing was able to schedule around 11% more task sets than the SA-based cache color assignment without re-sizing and around 13% more task sets than the full cache partitioning approach.

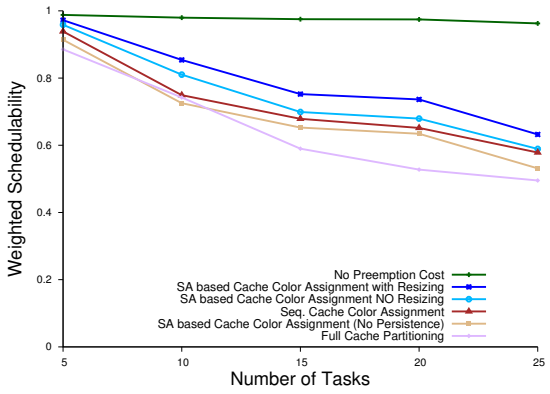
2) Number of Cache Colors: In this experiment, we evaluate the impact of cache size on the performance of the analyzed approaches by varying the number of cache color from 4 to 64. As the size of cache colors is constant (i.e., 512 B), increasing the number of cache colors also increases the cache size. All parameters other than the number of cache colors have the same values as used in the previous experiment. We have used the weighted schedulability measure defined by Bastoni et al. [5] to plot the results as shown in Fig. 5b. We observe that initially increasing the number of cache colors (i.e., from 4 to 8) decreases the schedulability of all approaches except the full cache partitioning approach. This is mainly because in this interval most cache colors were shared between tasks resulting in higher inter-task cache interference. However, even in this interval the SA-based cache color assignment with re-sizing outperforms all other approaches. A further increase in the number of cache colors results in reducing the number of cache colors that are shared among tasks. Therefore, we see an increase in the schedulability of all approaches. Understandably, the performance of the full cache partitioning approach is almost linear w.r.t the number of cache colors. Moreover, when the number of cache colors is large (e.g., 64) all approaches have similar results due to low cache interference.

3) Number of cache sets per cache color: We also performed an experiment by increasing the of number cache sets per cache color whilst keeping the cache size constant. We varied the size of one cache color between 1 to 128 cache sets with all other parameters set to default values. The resulting plot of task set schedulability w.r.t the number of cache sets per cache color is shown in Fig. 6a. We observe that when the size of a cache color is smaller all approaches were able to schedule more task sets. This is mainly because a smaller cache color size results in a tighter bound on the CRPD/CPRO suffered by the tasks. Whereas, increasing the size of a cache color decrease the total number of cache colors, potentially increasing the number of shared cache colors and the CRPD/CPRO suffered by the tasks. This results in decreasing task set schedulability for all approaches. Note that since the full cache partitioning approach uses the number of cache sets rather than cache colors, its performance is not affected by the size of a cache color.

4) Number of Tasks: To analyze the performance of all approaches w.r.t the number of tasks, we varied the number of tasks from 5 to 25 with all other parameters set to the same values as used in the core utilization experiment. Fig. 6b shows the result of the experiment. We observe that schedulability for all approaches decreases as the number of tasks is increased. For the full cache partitioning approach this decrease in schedulability is due to an increase in intra-task cache interference, i.e., as the number of tasks increase, less cache colors can be assigned to each individual task potentially resulting in increasing its intra-task cache interference. For the other approaches, the reduction in schedulability is due an increase in inter-task cache interference due to sharing of cache colors between several tasks. However, we observe that the SA-based cache color assignment with re-sizing still achieves much higher schedulability than all the other approaches.



(a) Increasing number of cache sets per cache color



(b) Varying the number of tasks

Figure 6: Schedulability w.r.t number of cache sets per color and number of tasks

9 RELATED WORK

A wealth of publications have studied the problem of cache interference. Wilhelm et al. [29] detailed static analysis and measurement-based methods to bound the intra-task cache interference in order to have deterministic bounds on the WCET of tasks. However, the work in [29] only focuses on the WCET analysis and does not consider the inter-task cache interference. Altmeyer et al. [2] presented the ECB-union and multi-set based methods (ECB-Union Multi-set and UCB-Union Multi-set) that dominate the state-of-the-art approaches to bound the inter-task cache interference (i.e., CRPD). However, their methods does not account for cache persistence between different jobs of a task and hence result in pessimistic WCRT bounds. Rashid et al. [24] introduced the notion of cache persistence and presented methods to bound the inter-task cache interference considering both CRPD and CPRO. They also integrated their approach in the WCRT analysis for FPPS, showing significant improvements in the accuracy of the response time analysis. However, when calculating CRPD/CPRO they only considered a sequential layout of tasks in memory, potentially overestimating the inter-task cache interference. In other approaches, cache partitioning [3, 7, 9, 11, 15, 22, 30] has been proposed to reduce or to completely mitigate the inter-task cache interference. Partitioning techniques can be implemented either in hardware [16] or in software [22, 30] and require specialized hardware/software (i.e., OS

or compiler) support. However, most of these works only focus on the implementation of cache partitioning [16, 22, 30] rather than evaluating the effects of cache partitioning on task set schedulability. Kim et al. [15] showed that the cache partitioning approaches are subjected to the problem of limited number of cache partitions. The authors then proposed a cache management scheme to assign private and shared cache partitions to tasks. However, their approach did not account for the intra-task cache interference and the method used to calculate the penalties due to sharing of cache partitions between tasks resulted in pessimistic CRPD bounds since it does not consider the actual ECBs/UCBs of tasks. Altmeyer et al. [3, 4] presented a cache-partitioning algorithm that is optimal under certain cache-modeling assumptions. However, the authors concluded that the trade off between intra- and inter-task cache interference often favors sharing the cache rather than partitioning it. In other works, Busquets et al. [9] and Bui et al. [7] proposed hybrid cache partitioning, where a designated cache area was shared between those tasks that are not allocated private cache partitions. However, these approaches were also focused on the inter-task cache interference and result in higher values of CRPDs when a large number of tasks use the shared cache area/partition.

The only existing approaches we are aware of and that relates to the work done in this paper was presented by Gebhard and Altmeyer [10] and later improved by Lunniss et al. [21]. Gebhard and Altmeyer [10] proposed an approach to optimize task layout in memory to improve task set schedulability by minimizing the inter-task cache conflicts. Their approach showed that with different task layouts in memory inter-task cache interference can be significantly reduced. However, their approach to bound the CRPDs was pessimistic since all ECBs of a task were treated as UCBs. Lunniss et al. [21] later improved the work done in [10] by using a more tighter approach for CRPD calculation. They proposed a simulated annealing based approach to optimize task layout in memory to reduce the inter-task cache interference and effectively improve schedulability. It has been identified in [3, 4] that an optimized layout of tasks in memory outperforms an optimal cache partitioning approach. However, in contrast to the work presented in this paper the task layout optimization approach presented in [21] only considered the inter-task cache interference (i.e., only CRPD) and does not account for CPRO. Moreover, they do not allow re-sizing the cache space assigned to tasks and only change task placements in memory to reduce CRPD while allowing tasks an unconstrained use of cache. This unconstrained use of the cache result in higher CRPDs when higher priority tasks have large cache footprints.

10 CONCLUSION AND FUTURE WORK

In this work, we showed that intra- and inter-task cache interference can be interrelated and balancing their respective contribution to tasks WCRT may result in improving task set schedulability. We use a cache coloring approach to optimize task layout in memory such that the trade-off between intra- and inter-task cache interference can be balanced. We also showed how the intra- and inter-task cache interference can be bounded under a cache coloring approach. Lastly, a simulated annealing algorithm is proposed to optimize the cache color assignment to tasks by re-allocating and re-sizing the cache colors assigned to tasks such that the task set's schedulability is achieved. Experiments were performed by varying different parameters using values from the Mälardalen benchmarks. Experimental results show that the proposed SA based cache color assignment of tasks dominates the existing approaches used to optimize task layout in memory.

The work presented assumed a direct mapped cache, future work could include extending it to N-way set associative caches. We also aim to extent this analysis to shared cache in multicore platforms.

A SIMULATED ANNEALING ALGORITHM

Algorithm 1 Simulated annealing based algorithm to optimize cache color assignment of tasks

Input: task set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$; total cache colors k^{total}
Output: Cache color assignment $\{ck_1, ck_2, ..., ck_n\}$; $true$ if τ is schedulable and $false$ otherwise.

1: **for** $i \leftarrow 1$ to n **do**
2: $\{ck_i\} = \emptyset$
3: **end for**
4: AssignSequentialColors(τ,k^{total});
5: **if** isSchedulable(τ) **then**
6: **return** true;
7: **else**
8: SimulatedAnnealing(τ);
9: **if** isSchedulable(τ) **then**
10: **return** true;
11: **else**
12: **return** false;
13: **end if**
14: **end if**
15: **Function** SimulatedAnnealing(τ)
16: $CurrentTemp \leftarrow 400$;$DesiredTemp \leftarrow 0.001$; $CoolingRate \leftarrow 0.99$;
17: **while** $CurrentTemp \geq DesiredTemp$ **do**
18: $TaksetSlackOld = Calculatetasksetslack(\tau)$;
19: $SelectRandom(ReAllocate(), ShiftLayout(), ReSize())$;
20: $TaksetSlackNew = Calculatetasksetslack(\tau)$;
21: ΔSlack$\leftarrow TaksetSlackOld - TaksetSlackNew$
22: **if** $\Delta Slack \geq 0$ **then**
23: Accept new cache color assignment of τ;
24: **else**
25: $Randomprob \leftarrow rand(0, 1)$
26: **if** $Randomprob < e^{\frac{-\Delta Slack}{CurrentTemp}}$ **then**
27: Accept new cache color assignment of τ;
28: **else**
29: Discard new cache color assignment of τ;
30: **end if**
31: **end if**
32: $CurrentTemp = CurrentTemp * CoolingRate$;
33: **end while**
34: **end function**

ACKNOWLEDGMENTS

This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (CEC/04234); also by FCT and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/119150/2016.

REFERENCES

[1] S. Altmeyer, R. I. Davis, and C. Maiza. 2011. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *RTSS*. 261–271.

[2] 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.

[3] S. Altmeyer, R. Douma, W. Lunniss, and R.I. Davis. 2014. Evaluation of Cache Partitioning for Hard Real-Time Systems. In *ECRTS*. 15–26.

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

[6] E. Bini and G. C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1-2 (2005), 129–154.

[7] Bach D Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. 2008. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA*. IEEE, 101–110.

[8] J. 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 *RTAS*. 204–212.

[9] José V Busquets-Mataix, Juan José Serrano, and Andy Wellings. 1997. Hybrid instruction cache partitioning for preemptive real-time systems. In *Real-Time Systems, Proceedings., Ninth Euromicro Workshop on*. IEEE, 56–63.

[10] Gernot Gebhard and Sebastian Altmeyer. 2007. Optimal task placement to improve cache performance. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*. ACM, 259–268.

[11] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 32.

[12] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, Vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[13] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[14] Mathai Joseph and P Pandya. 1986. Finding response times in a real-time system. *Comput. J.* 29, 5 (1986), 390–395.

[15] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. 2013. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS*. IEEE, 80–89.

[16] David B Kirk and Jay K Strosnider. 1990. SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In *RTSS*. IEEE, 322–330.

[17] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[18] C. G. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *Computers, IEEE Transactions on* 47, 6 (1998), 700–713.

[19] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *RTAS*. IEEE, 213–224.

[20] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*. IEEE, 367–378.

[21] W. Lunniss, S. Altmeyer, and R.I. Davis. 2012. Optimising Task Layout to Increase Schedulability via Reduced Cache Related Pre-emption Delays. In *RTNS*. 161–170.

[22] Frank Mueller. 1995. Compiler support for software-based cache partitioning. In *ACM Sigplan Notices*, Vol. 30. ACM, 125–133.

[23] Syed Aftab Rashid, Geoffrey Nelissen, Sebastian Altmeyer, Robert I Davis, and Eduardo Tovar. 2017. Integrated Analysis of Cache Related Preemption Delays and Cache Persistence Reload Overheads. In *RTSS*. IEEE, 188–198.

[24] S. A. Rashid, G. Nelissen, D. Hardy, B. Akesson, I. Puaut, and E. Tovar. 2016. Cache-Persistence-Aware Response-Time Analysis for Fixed-Priority Preemptive Systems. In *ECRTS*. 262–272.

[25] J. Staschulat, S. Schliecker, and R. Ernst. 2005. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS*. 41–48.

[26] Vivy Suhendra and Tulika Mitra. 2008. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*. ACM, 300–303.

[27] Y. Tan and V. Mooney. 2007. Timing analysis for preemptive multitasking real-time systems with caches. *ACM TECS* 6, 1 (2007), 7.

[28] H. Tomiyama and N. D. Dutt. 2000. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*. 67–71.

[29] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problem: overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 36.

[30] Andrew Wolfe. 1993. Software-based cache partitioning for real-time applications. In *Third International Workshop on Responsive Computer Systems*.

[31] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 89–102.