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 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 are 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, 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.
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 $\tau_1$ and task $\tau_2$ 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).
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 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. The cache is assumed to be direct-mapped with \( k_{\text{total}} \) colors. Each color is uniquely numbered between 1 to \( k_{\text{total}} \). The size of a cache color is denoted by \( k_{\text{size}} \) 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.

We consider a fixed priority scheduling (e.g., Rate Monotonic or Deadline Monotonic) of a set of sporadic tasks. The task set \( \tau \) comprises \( n \) tasks, i.e., \( \tau = \{ \tau_1, \ldots, \tau_n \} \). Each task \( \tau_i \) is defined by a triplet \( (C_i[k_i], T_i, D_i) \), where \( C_i[k_i] \) is a vector of length \( k_{\text{total}} \) that contains the worst-case execution time of task \( \tau_i \) in isolation assuming \( k_i \) contiguous cache colors are assigned to \( \tau_i \). Note that \( k_i \) represents the number of cache colors used by \( \tau_i \), whereas the set of cache colors assigned to \( \tau_i \) is denoted by \( c_k \). The minimum inter-arrival time of \( \tau_i \) 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 \( \tau_i \) 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 \( \tau_i \) executing in isolation and assuming that \( k_i \) contiguous cache colors are assigned to \( \tau_i \). It is usually assumed that \( C_i[k_i] \) is non-increasing with \( k_i \), i.e., \( k_i < k_{i+1} \Rightarrow C_i[k_i] \geq C_i[k_{i+1}] \). However, we note that since \( PD_i \) is independent of the number of cache colors assigned to \( \tau_i \), 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 \( \tau_i \), i.e., \( k_i < k_{i+1} \Rightarrow MD_i[k_i] \geq MD_i[k_{i+1}] \). Note that \( PD_i \) and \( MD_i[k_i] \) may not necessarily be

experienced on the same execution path of \( \tau_i \). 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.

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

3 BACKGROUND

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

\[ R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \frac{R_j}{T_j} \times C_j \]  

Eq. (1) uses the worst-case execution time (WCET) of tasks in isolation to account for the interference task \( \tau_j \) may suffer due to preemptions by all higher priority tasks in \( hp(i) \). 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 \( \tau_i \) due to preemptions by higher priority tasks in \( hp(i) \), e.g., when a lower priority task \( \tau_i \) is preempted by a higher priority task \( \tau_j \in \text{hp}(i) \), the preempting task \( \tau_j \) may evict cache blocks of the preempted task \( \tau_i \) that has to be reloaded after task \( \tau_j \) resumes its execution. These extra cache reloads during the execution of task \( \tau_i \) are termed as CRPDs. CRPD of task \( \tau_i \) 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

\[ R_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left( \frac{R_j}{T_j} \times (C_j + \gamma_{i,j}) \right) \]  

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 \( \tau_i \). 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.

\footnote{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.}

\footnote{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.}

\footnote{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 $\tau_i$ 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 $\text{hp}(i) \setminus \tau_i$. CPRO of a higher priority task $\tau_j \in \text{hp}(i)$ executing during the response time of a lower priority task $\tau_i$ is usually denoted by $\rho_{j,i}$ and is formally defined as [24] "the maximum memory reload overhead suffered by a task $\tau_j$ due to evictions of its persistence cache blocks (PCBs) by tasks in $\text{hp}(i) \setminus \tau_j$ while $\tau_j$ is executing during the response time of $\tau_i$". 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 $\tau_i$ were loaded in the cache by a previous job of $\tau_i$, the memory demand of subsequent jobs of $\tau_i$ can be much lower than the worst-case memory demand of $\tau_i$ in isolation. This type of memory demand is called the residual memory demand of $\tau_i$ 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". In this work, we propose a cache coloring approach where the residual memory demand of task $\tau_i$ also depends on the number of cache colors assigned to $\tau_i$, i.e., $k_i$. Hence, in the remaining of the paper we will consider the residual memory demand of task $\tau_i$ by $\text{MD}^i_r[k_i]$. Effectively, the total memory demand $\text{MD}(t)$ of a task $\tau_i$ within a time window of length $t$ when $\tau_i$ executes in isolation is defined as [23, 24]

$$\text{MD}(t) = \min \left( \frac{t}{T_j} \times \text{MD}^i_r[k_i], \frac{t}{T_j} \times \text{MD}^i_r[k_i] + |\text{PCB}_i| + d_{\text{mem}} \right)$$

(3)

Furthermore, Rashid et al. [23, 24] also presented the WCRT analysis for FPPs that accounts for the intra-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 $\tau_i$ is calculated in [23, 24] as

$$R_i = C_i + \sum_{\forall j \text{ job}(i)} \left( \frac{R_j}{T_j} \gamma_{i,j} + \min \left( \frac{R_j}{T_j} C_j, \frac{R_j}{T_j} PD_j + \text{MD}(R_i) + \rho_{j,i} \right) \right)$$

(4)

where $\gamma_{i,j}$ and $\rho_{j,i}$ bounds the CRPD and CPRO considering the pair of tasks $\tau_i$ and $\tau_j$ 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 $\tau_i$ 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 $\tau_i$, i.e.,

$$R_i = C_i^\text{min} + C_i^\text{intrak}_i + \sum_{\forall j \text{ job}(i)} \min \left( \frac{R_i}{T_j} \left( C_i^\text{min} + C_i^\text{intrak}_i \right), \frac{R_i}{T_j} PD_j \right)$$

(5)

$$+ \text{MD}(R_i) + C_i^\text{intrar}_i(R_i) + C_i^\text{intrar}_j(R_i)$$

Figure 1: Increase in execution and memory demand of task $\tau_i$ due to reduction in number of cache colors assigned to $\tau_i$. In Eq. (5), $C_i^\text{min}$ denotes the worst-case execution time of task $\tau_i$ in isolation assuming $\tau_i$ is allocated a cache of infinite size (or more practically, the total cache space assigned to task $\tau_i$ is greater or equal to the size of $\tau_i$ in main memory). The intra-task cache interference of $\tau_i$ w.r.t the number of cache colors $k_i$ assigned to $\tau_i$ is denoted by $C_i^\text{intrak}_i$, as intra-task interference impacts only the execution time of $\tau_i$ itself. Similarly, the intra-task cache interference $C_i^\text{intrar}_i$ of each higher priority task $\tau_j \in \text{hp}(i)$ executing during the response time of $\tau_i$ is considered in the higher priority interference term within the sum on higher priority tasks. Moreover, $C_i^\text{intrar}_j(R_i)$ denotes the inter-task cache interference in terms of CRPD that task $\tau_j$ may suffer during its response time due to preemptions by all higher priority tasks in $\text{hp}(i)$ and $C_i^\text{intrar}_j(R_i)$ 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 $\tau_i$. Note that $\text{MD}(R_i)$ in Eq. (5) is calculated in a similar manner to Eq. (4) (i.e., using Eq. (3)). Since $\text{MD}(R_i)$ is a function of $\text{MD}^i_r[k_j]$, $\text{MD}^j_r[k_j]$ and the number of PCBs of $\tau_j$, which are in turn functions of the number of cache colors $k_j$ assigned to $\tau_j$ therefore, $\text{MD}(R_i)$ directly considers the intra-task interference of all jobs of $\tau_j$ executing during the response time of $\tau_i$.

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 $\tau_i$ its intra-task cache interference depends on the cache space or the number of cache colors $k_i$ assigned to $\tau_i$. Consider the plot of worst-case execution time ($C_i^\text{min}$) and the worst-case memory demand (MD($k_i$)) of task $\tau_i$ with respect to the number of cache colors $k_i$ assigned to $\tau_i$ 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 fdt 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 $\tau_1$ is greater or equal to the size of $\tau_1$ in main memory (i.e., for $k_1 \geq 4$), the worst-case execution time ($C_i[k_1]$) and the worst-case memory demand ($MD_i[k_1]$) of $\tau_1$ is minimum, i.e., $C_i[k_1] = C_i^{\text{min}}$ and $MD_i[k_1] = MD_i^{\text{min}}$ for $k_1 \geq 4$, where $MD_i^{\text{min}}$ represents the worst-case memory access demand of task $\tau_1$ in isolation assuming $\tau_1$ is allocated an infinite cache size. Effectively, for $k_1 \geq 4 \tau_1$ 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 $k_1$ assigned to $\tau_1$, its worst-case execution time ($C_i[k_1]$) and the worst-case memory demand ($MD_i[k_1]$) tend to increase. This increase in $C_i[k_1]$ and $MD_i[k_1]$ is due to an increase in the intra-task cache interference of $\tau_1$ mainly because by reducing the number cache colors $k_1$, the number of UCBs of task $\tau_1$ may also decrease, i.e., by decreasing the number of cache colors $k_1$ (or the cache space) assigned to $\tau_1$, cache blocks of $\tau_1$ 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 $\tau_1$ that can be reused, i.e., the number of UCBS. Effectively, this reduction of the number of UCBS results in increasing $MD_i[k_1]$ of $\tau_1$ for $k_1 < 4$. Therefore, the intra-task cache interference of a task directly relates to its worst-case memory access demand in the following manner:

$$C_i^{\text{intra},k_1} = MD_i[k_1] - MD_i^{\text{min}}$$

(6)

The resulting intra-task cache interference of $\tau_1$ for a given cache color assignment $k_1$, i.e., $C_i^{\text{intra},k_1}$, is accounted for in the WCRT of $\tau_1$ (i.e., Eq. (5)) by explicitly adding $C_i^{\text{intra},k_1}$ to $C_i^{\text{min}}$ which is the worst-case execution time of $\tau_1$ in isolation assuming $\tau_1$ is allocated an infinite cache. However, we note that because $C_i^{\text{intra},k_1}$ depends on $MD_i[k_1]$ and since $MD_i[k_1]$ may not necessarily be experienced on the same execution path of $\tau_1$ for different cache color assignments $k_1$, it holds that $C_i[k_1] \leq C_i^{\text{min}} + C_i^{\text{intra},k_1}$. Hence, Eq. (6) provides a safe upper-bound on intra-task cache interference even for multi-path programs.

6 INTER-TASK CACHE INTERFERENCE

Under FPPS, the inter-task cache interference a task $\tau_j$ may suffer due to higher priority tasks in hpi($i$) 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 $\tau_1$ due to preemptions by a higher priority task $\tau_j$ in hpi($i$) and is denoted by $C_i^{\text{inter},p}$. Whereas, the inter-task cache interference in terms of CPRO results from the eviction of PCBs of the higher priority task $\tau_j$ in hpi($i$) due to the executions of all other tasks in the system (while $\tau_j$ executes during the response time of $\tau_j$) and is denoted by $C_i^{\text{inter},p}$. In the following subsections, we explain how $C_i^{\text{inter},p}$ and $C_i^{\text{inter},p}$ can be bounded under the 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 $Y_{i,j}$ 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 $\tau_j$ and the UCBS of all tasks in aff($i,j$) possibly affected by the preemption caused by $\tau_j$ to calculate $Y_{i,j}$. Formally,

$$Y_{i,j} = d_{\text{mem}} \times \left(\bigcup_{s \in \text{aff}(i,j)} \text{UCB}_s \cap \text{ECB}_j\right)$$

(7)

where, $\text{ECB}_j$ and $\text{UCB}_s$ are the sets of ECBs and UCBS of task $\tau_j$ and $\tau_s$, 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 $\tau$. Indeed, the actual mapping of tasks within a cache color may not be known as it is handled by the cache controller. Consequently, the actual set of UCBS/ECBS of tasks may not be known as they depend on the actual cache sets used by the tasks. For example, consider two tasks $\tau_1$ and $\tau_2$ sharing the same cache color $c_k$ where $c_k$ comprises 4 cache sets, numbered from 1 to 4. If both $\tau_1$ and $\tau_2$ have 2 UCBs under this cache assignment, these UCBs can be mapped to any of the four cache sets depending on how $\tau_1$ and $\tau_2$ are mapped within $c_k$ by the cache controller, i.e., $\text{UCB}_1 = \{1, 2\}$ and $\text{UCB}_2 = \{3, 4\}$ or any other combinations with or without overlapping between $\text{UCB}_1$ and $\text{UCB}_2$. 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 $Y_{i,j}$.

In order to bound the CRPD $Y_{i,j}$ under our cache coloring approach, we first determine the cache colors that may be affected when $\tau_j$ is preempted by a higher priority task $\tau_j \in \text{hpi}(i)$. Assuming that the cache color assignment of tasks has already been done, i.e., $\tau_1$ and $\tau_j$ are assigned a set of $c_k$ and $c_k$ cache colors respectively. We know from the UCB-union approach (Eq. (7)), that when a task $\tau_j$ is preempted by a higher priority task $\tau_j$, the set of UCBS of all tasks in aff($i,j$) can be evicted. Similarly, when a task $\tau_j$ using a set of $c_k$ cache colors is preempted by a higher priority task $\tau_j$, whose assigned a set of $c_k$ cache colors, the cache colors used by all tasks in aff($i,j$) may be evicted. Therefore, the maximum number of cache colors that may be affected due to a single preemption of $\tau_j$ by $\tau_j$ is bounded by $k_{i,j}$, where

$$k_{i,j} = \left|\bigcup_{s \in \text{aff}(i,j)} \text{ck}_s \cap \text{ck}_j\right|$$

(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 $\tau_j$ by $\tau_j$. Therefore, the product $k_{i,j} \times k_{\text{size}}$ can be used to upper bound the number of cache sets that may be evicted due to a single preemption of $\tau_j$ by $\tau_j$. However, this bound can obviously be very pessimistic, mainly because it does not consider the actual number of UCBS in

Footnote: 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\(_j(k_j)\) be defined as

- \( \text{UCB}_i(k_i) \): The maximum number of UCBs of task \( \tau_i \) when it is assigned \( k_i \) cache colors.
- \( \text{ECB}_j(k_j) \): The maximum number of ECBs of task \( \tau_j \) when it is assigned \( k_j \) cache colors.

Effectively, the CRPD cost due to a single preemption of \( \tau_j \) by \( \tau_j \) can be bounded using the notion of UCB\(_i(k_i)\) and ECB\(_j(k_j)\).

### Lemma 1
The CRPD cost due to a single preemption of a lower priority task \( \tau_i \) by a higher priority task \( \tau_j \) is bounded by \( y_{i,j}^{col} \), i.e.,

\[
y_{i,j}^{col} = d_{mem} \times \min \left\{ \sum_{V \in \text{aff}(i,j)} (\text{UCB}_i(k_i) \times V_{s,j}) ; \text{ECB}_j(k_j) \right\}
\]

where \( V_{s,j} = 1 \) if \( |c_k \cap c_j| > 0 \) and \( V_{s,j} = 0 \), otherwise.

**Proof.** We prove that both \( \sum_{V \in \text{aff}(i,j)} (\text{UCB}_i(k_i) \times V_{s,j}) \) and \( \text{ECB}_j(k_j) \) are upper bounds on the CRPD cost \( y_{i,j}^{col} \). Therefore, the minimum between the two is also an upper bound on \( y_{i,j}^{col} \).

1. From the UCB-union approach (Eq. (7)), it follows that when task \( \tau_j \) is preempted by a higher priority task \( \tau_i \), the set of UCBs of all tasks in \( \text{aff}(i,j) \) may be evicted. However, when using cache coloring the actual set of UCBs of a task \( \tau_k \in \text{aff}(i,j) \) may not be known. Instead, we know the maximum number of UCBs of \( \tau_k \), i.e., \( \text{UCB}_i(k_i) \), for a given cache color assignment \( c_k \) with size \( k_i \). Also due to cache coloring, \( \tau_j \) can only evict UCBs of a task \( \tau_k \in \text{aff}(i,j) \) only when \( |c_k \cap c_j| > 0 \) (i.e., \( V_{s,j} = 1 \)). Hence, the total number of UCBs among all tasks in \( \text{aff}(i,j) \) that can be evicted by \( \tau_j \) is bounded by \( \sum_{V \in \text{aff}(i,j)} (\text{UCB}_i(k_i) \times V_{s,j}) \). Therefore, for a single preemption of \( \tau_j \) by \( \tau_j \), \( \sum_{V \in \text{aff}(i,j)} (\text{UCB}_i(k_i) \times V_{s,j}) \) upper bounds the CRPD cost \( y_{i,j}^{col} \).

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 \( \tau_i \) by \( \tau_j \) the number of ECBs of \( \tau_j \) also upper bounds the CRPD cost. However, due to cache coloring not all cache colors used by \( \tau_j \), i.e., \( k_j \), may overlap with cache colors used by task \( \tau_j \) (and by tasks in \( \text{aff}(i,j) \)) except for \( k_{i,j} \) cache colors (i.e., Eq. (8)).

Hence, the maximum number of ECBs of \( \tau_j \) in the \( k_{i,j} \) overlapping cache colors used by tasks in \( \text{aff}(i,j) \), i.e., \( \text{ECB}_j(k_{i,j}) \), upper bounds the CRPD cost \( y_{i,j}^{col} \) from \( \tau_j \)'s perspective.

The lemma follows.

For a single preemption of \( \tau_j \) by \( \tau_j \), the CRPD cost can be bounded using Lemma 1. However as we will now prove, the actual time taken to reload all UCBs of \( \tau_j \) from the main memory is also bounded

\[ \text{Reduction in time to access main memory due to an increase in the number of cache colors (or UCBs) of } \tau_j \]

![Figure 2: Worst-case memory demand \( MD_i(k_i) \) of task \( \tau_i \) w.r.t. the number of cache colors assigned to \( \tau_i \).](image)

By the change in the worst-case memory demand of task \( \tau_i \) w.r.t. the number of cache colors \( k_i \) assigned to \( \tau_i \).

To illustrate, let \( MD_i^{max} \) be the maximum worst-case memory demand of \( \tau_i \) when there is no cache assigned to \( \tau_i \) (i.e., \( k_i = 0 \)). Now, consider the example plot of main memory access demand \( MD_i(k_i) \) of a task \( \tau_i \) shown in Fig. 2. The plot shows the normalized worst-case memory access demand of the fcft 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 \( \tau_i \) is maximum, i.e., \( MD_i(k_i) = MD_i^{max} \). Also, for \( k_i = 0 \) since no cache space is assigned to \( \tau_i \) there cannot be any useful cache blocks, i.e., \( UCB_j = \emptyset \). 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 \( \tau_i \) 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 \( \tau_i \), more instructions/data of \( \tau_i \) may remain cached and therefore reused without having to reload them from main memory. This effectively increases the number of UCBs of \( \tau_i \), leading to a reduction in its worst-case memory access demand. The change in the worst-case memory demand \( MD_i(k_i) \) of \( \tau_i \) due to an increase in the number of cache colors \( k_i \) assigned to \( \tau_i \) can be bounded by \( \Delta MD_i(k_i) \), where

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

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

\[
\text{UCB}_i(k_i) \times N_i \times d_{mem} \leq \Delta MD_i(k_i)
\]

where \( N_i \) is the average number of times each UCB of \( \tau_i \) is accessed while it is cached.

Since \( \Delta MD_i(k_i) \) bounds the time to reload all UCBs of \( \tau_i \) for a given cache color assignment \( k_i \), it also bounds the total CRPD \( \tau_j \) can suffer due to eviction of its UCBs by tasks in \( hp(i) \). However, we know from Lemma 1 that when task \( \tau_j \) is preempted by a higher priority task \( \tau_j \in hp(i) \), UCBs of all tasks in \( \text{aff}(i,j) \) can be evicted. Therefore, to bound the total CRPD \( \tau_j \) 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 \( \text{aff}(i,j) \) should be considered.
Lemma 2. The total CRPD cost suffered by a task \( \tau_j \) due to pre-emptions by a higher priority task \( \tau_i \in \text{hp}(i) \) is bounded by

\[
\forall j \in \text{hp}(i) : y^{\text{tot}}_{j,i} \leq \sum_{\forall v \in \text{aff}(i,j)} \Delta M D_{k}\{k_s\} \tag{12}
\]

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

Base Case: Consider two tasks \( \tau_i \) and \( \tau_{i-1} \) such that \( \tau_{i-1} \) is a priority just above that of \( \tau_i \). Therefore, \( \text{aff}(i, i-1) = \text{true} \).

The total CRPD that \( \tau_i \) may suffer due to \( \tau_{i-1} \) is the time window of length \( t \) during which \( \tau_i \) is affected by \( \tau_{i-1} \)’s cache operations.

Induction step: Consider another task \( \tau_{s} \) having a priority higher than \( \tau_i \) and assume that Eq. (12) holds for \( j = s \), then Eq. (12) also holds for \( j = s - 1 \).

Since a higher priority task \( \tau_s \in \text{hp}(i) \) can release \( t_{j} \) jobs during a time window of length \( t \) and the CRPD caused by each of these jobs on \( \tau_i \) can be calculated by Eq. (9), therefore the total CRPD that \( \tau_i \) may suffer due to \( \tau_s \) is bounded by \( y^{\text{col}}_{j,i} \).

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

\[
c^{\text{inter},Y}(t) = \left( \frac{t}{t_{j}} \times y^{\text{col}}_{j,i} \times \sum_{\forall v \in \text{aff}(i,j)} \Delta M D_{k}\{k_s\} \right) \tag{13}
\]

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 \( \tau_i \), the CPRO-union approach makes use of the set of PCBs of task \( \tau_j \) and the set of ECBs of all tasks in \( \text{hp}(i) \) \( \setminus \tau_j \), i.e.,

\[
\rho_{j,i} = d_{\text{mem}} \times \left( \sum_{\forall v \in \text{aff}(i,j)} \text{PCB}_j \cap \bigcup_{\tau_s \in \text{hp}(i) \setminus \tau_j} \text{ECB}_s \right) \tag{14}
\]

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 \( \tau_j \) executing during the response time of \( \tau_i \), 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 successive jobs of \( \tau_j \). Assuming \( \tau_j \) and \( \tau_i \) are assigned a set of \( c_k \) and \( c_{k'} \) cache colors respectively, then the maximum number of cache colors of \( \tau_j \) that can be evicted between its successive jobs due to the executions of all tasks in \( \text{hp}(i) \setminus \tau_j \) during the response time of \( \tau_i \) can be bounded by \( k'_{j,i} \), calculated as follows.

\[
k'_{j,i} = \left\lfloor \frac{c_{k'} - \text{PCB}_j \cap \bigcup_{\forall v \in \text{aff}(i,j)} \text{ECB}_s}{c_k} \right\rfloor \tag{15}
\]

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

However, this is obviously pessimistic and to have a tighter bound on the CPRO in terms of the number of PCBs of \( \tau_j \) that may be evicted between its two successive jobs, we define \( \text{PCB}(k) \), i.e.,

- \( \text{PCB}(k) \): The maximum number of PCBs of task \( \tau_i \) when it is assigned \( k \) cache colors.

\( \text{PCB}(k) \) can also be computed in a similar manner to \( \text{ECB}(k) \) and \( \text{UCB}(k) \) as detailed in Section 6.1. Furthermore, \( \text{PCB}(k) \) and \( \text{ECB}(k) \) can be used to bound the CPRO of a task \( \tau_j \in \text{hp}(i) \) executing during the response time of \( \tau_i \) using the following lemma

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

\[
\rho_{j,i}^{\text{col}} = \min \left( \text{PCB}(k) \setminus \bigcup_{\forall v \in \text{aff}(i,j)} \text{ECB}(k_{j,i}'), \text{ECB}(k_{j,i}') \times \text{PCB}(k) \right) \tag{16}
\]

where \( k_{j,i}' \) is the number of cache colors that may overlap and potentially evict PCBs of \( \tau_j \).
Eq. (16) implies that the CPRO of one job of task \( \tau_j \) is upper bounded by the difference between the worst-case and the residual memory demand of \( \tau_j \), i.e.,

\[
\rho_{j,1}^{one} \leq \Delta M D_j[k_j] = \Delta M D_j[k_j] - \Delta M D_j[k_j - 1]
\]

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 \( \tau_j \) executing during the response time of a task \( \tau_i \). However, we know that task \( \tau_i \) may execute several times during the execution of \( \tau_j \). Therefore, the total inter-task cache interference in terms of CPRO suffered by \( \tau_j \) while executing during the response time of \( \tau_i \) 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 the cache 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

\[
C_{j,i}^{inter,p} = \left( \frac{t}{T_j} - 1 \right) \times \min \left( \rho_{j,1}^{col}, \Delta M D_j[k_j] \right)
\]

Proof. (1) It is proved in [23] that in a time interval of length \( t \) at most \( \left\lceil \frac{t}{T_j} \right\rceil - 1 \) jobs of task \( \tau_j \) can suffer CPRO. (2) It implies that both \( \rho_{j,1}^{col} \) (Eq. (16)) and \( \Delta M D_j[k_j] \) (by Lemma 4 and Eq. (17)) upper bound the CPRO suffered by one job of \( \tau_j \) executing during the response time of \( \tau_i \). Therefore, the minimum between the two bounds is also an upper bound on the CPRO suffered by a single job of \( \tau_j \) during the response time of \( \tau_i \).

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 \( \tau \). 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 \( \tau \). If all tasks in task set \( \tau \) are schedulable with the sequential cache color allocation (i.e., \( \tau \) is schedulable), no changes are made to the cache color assignment of tasks and the algorithm returns true and exit. However, if \( \tau \) 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. 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 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 $\tau$ 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 $t_i$ is denoted by $S_i$ and is defined as \( \text{the difference between the relative deadline and the WCRT of } t_i \), i.e., $S_i = D_i - R_i$, where $R_i$ is calculated by considering the worst-case interference on $t_i$ by all higher priority tasks in $\text{hp}(i)$, i.e., by setting $R_i = \tau_1$ in Equation (5). The total slack $S_{\text{tot}}$ of task set $\tau$ is given as

\[
S_{\text{tot}} = \sum_{i=1}^{n} w_i \times S_i
\]  

(21)

where $w_i$ is the weight assigned to every $t_i \in \tau$ such that, $w_i = 0$ if $S_i \geq 0$ and $w_i = 1$, 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., $e^{-\frac{\Delta \text{Slack}_{\text{new}}}{k \cdot \text{temp}}}$ (see Appendix A), then the new cache color assignment for $\tau$ 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., $r_1 = \text{minmax}$ to $r_{10} = \text{bsort100}$, where $r_1$ 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 $k_i = \left\lceil \frac{ECB(k_i)}{100} \right\rceil$.

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., $k_{\text{total}} = 32$ with each color having a size of 512 Bytes, i.e., $k_{\text{size}} = 512$ Bytes. The block reload time $d_{\text{mem}}$ was set to 10µs.

#### Table 1: task set parameters used in the working example

<table>
<thead>
<tr>
<th>Name</th>
<th>$C_i[k_i]$</th>
<th>$PD_i$</th>
<th>$MD_i[k_i]$</th>
<th>$k_i$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>minmax</td>
<td>2522</td>
<td>122</td>
<td>2400</td>
<td>2</td>
<td>14315</td>
</tr>
<tr>
<td>lcdnum</td>
<td>3440</td>
<td>984</td>
<td>2740</td>
<td>2</td>
<td>73143</td>
</tr>
<tr>
<td>cnt</td>
<td>10090</td>
<td>7191</td>
<td>3818</td>
<td>2</td>
<td>85816</td>
</tr>
<tr>
<td>ns</td>
<td>30149</td>
<td>28149</td>
<td>9172</td>
<td>2</td>
<td>109744</td>
</tr>
<tr>
<td>statemen</td>
<td>43344</td>
<td>10586</td>
<td>35297</td>
<td>18</td>
<td>636613</td>
</tr>
<tr>
<td>insertsort</td>
<td>7574</td>
<td>5974</td>
<td>2343</td>
<td>1</td>
<td>734873</td>
</tr>
<tr>
<td>nischneu</td>
<td>316409</td>
<td>22009</td>
<td>294000</td>
<td>38</td>
<td>1889824</td>
</tr>
<tr>
<td>qsort</td>
<td>26141</td>
<td>9241</td>
<td>77713</td>
<td>3</td>
<td>2899034</td>
</tr>
<tr>
<td>IR</td>
<td>1575800</td>
<td>123681</td>
<td>45816</td>
<td>9</td>
<td>6250339</td>
</tr>
<tr>
<td>bsort100</td>
<td>712289</td>
<td>710289</td>
<td>90893</td>
<td>2</td>
<td>26727122</td>
</tr>
</tbody>
</table>

The task set was created by fixing the core utilization at 0.8, with task utilisations 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 initialize 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
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 \( \tau_7 \) on all lower priority tasks (i.e., \( \tau_8, \tau_9 \) and \( \tau_{10} \)) was reduced by increasing the intra-task cache interference of \( \tau_7 \) (i.e., by reducing the number of cache colors used by \( \tau_7 \)). 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 \( C_i[k_i], PD_i, MD_i[k_i], MD'_i[k_i], UCB_i(k_i), ECB_i(k_i) \) 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 \( k_i = \left\lceil \frac{ECB_i(k_i)}{U_i} \right\rceil \). Each task was randomly assigned the values \( C_i[k_i], PD_i, MD_i[k_i], MD'_i[k_i], UCB_i(k_i), ECB_i(k_i), PCB_i(k_i) \) 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 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/re-sizning 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 utilization, 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 number of 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.

![Figure 6: Schedulability w.r.t number of cache sets per color and number of tasks](image)

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.
We use a cache coloring approach to optimize task layout in memory. 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 that 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 Luniss 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. Luniss 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 \( \{ c_{k1}, c_{k2}, ..., c_{kn} \} \); true if \( \tau \) is schedulable and false otherwise.

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


