CITTA: Cache Interference-aware Task Partitioning for Real-time Multi-core Systems

Jun Xiao
University of Amsterdam
Amsterdam, Netherlands
J.Xiao@uva.nl

Andy D. Pimentel
University of Amsterdam
Amsterdam, Netherlands
A.D.Pimentel@uva.nl

Abstract

Shared caches in multi-core processors introduce serious difficulties in providing guarantees on the real-time properties of embedded software due to the interaction and the resulting contention in the shared caches. Prior work has studied the schedulability analysis of global scheduling for real-time multi-core systems with shared caches. This paper considers another common scheduling paradigm: partitioned scheduling in the presence of shared cache interference. To achieve this, we propose CITTA, a cache-interference aware task partitioning algorithm. An integer programming formulation is constructed to calculate the upper bound on cache interference exhibited by a task, which is required by CITTA. We conduct schedulability analysis of CITTA and formally prove its correctness. A set of experiments is performed to evaluate the schedulability performance of CITTA against global EDF scheduling over randomly generated tasksets. Our empirical evaluations show that CITTA outperforms global EDF scheduling in terms of task sets deemed schedulable.

CCS Concepts • **Computer systems organization → Embedded software**.

Keywords Shared caches, Partitioned scheduling, Schedulability analysis, Real-time systems

ACM Reference Format:
Jun Xiao and Andy D. Pimentel. 2020. CITTA: Cache Interference-aware Task Partitioning for Real-time Multi-core Systems. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '20), June 16, 2020, London, United Kingdom.* ACM, London, UK, 11 pages. https://doi.org/10.1145/3372799.3394367

LCTES '20, June 16, 2020, London, United Kingdom

ACM ISBN 978-1-4503-7094-3/20/06…$15.00
https://doi.org/10.1145/3372799.3394367

1 Introduction and Motivation

Caches are common on multi-core systems as they can efficiently bridge the performance gap between memory and processor speeds. The last-level caches are usually shared by cores to improve utilization. However, this brings major difficulties in providing guarantees on real-time properties of embedded software due to the interaction and the resulting contention in a shared cache.

On a multi-core processor with shared caches, a real-time task may suffer from two different kinds of cache interferences [21], which severely degrade the timing predictability of multi-core systems. The first is called intra-core cache interference, which occurs within a core, when a task is preempted and its data is evicted from the cache by other real-time tasks. The second is inter-core cache interference, which happens when tasks executing on different cores access the shared cache simultaneously. In this work, we consider non-preemptive task systems, which implies that intra-core cache interference is avoided since no preemption is possible during task execution. We therefore focus on inter-core cache interference.

It is necessary to conduct schedulability analysis when designing hard real-time application systems executing on multi-core platforms with shared caches, as those systems cannot afford to miss deadlines and hence demand timing predictability. Any schedulability analysis requires knowledge about the Worst-Case Execution Time (WCET) of real-time tasks. However, as pointed out in [28], it is extremely difficult to predict the cache behavior to accurately obtain the WCET of a real-time task considering cache interference since different cache behaviors (cache hit or miss) will result in different execution times of each instruction. In this paper, we assume that a task's WCET itself does not account for shared cache interference but, instead, we determine this interference explicitly (as will be explained later on). Hardy and Puaut [18] present such an approach to derive a task's WCET without considering shared cache interference.

On multi-core systems, two paradigms are widely used for scheduling real-time tasks: global and partitioned (semi-partitioned) scheduling. For global scheduling, a job is allowed to execute on any core. In partitioned scheduling, on the other hand, tasks are statically allocated to processor cores, i.e., each task is assigned to a core and is always executed on that particular core. Although the partitioned

approaches cannot exploit all unused processing capacity since a bin-packing-like problem needs to be solved to assign tasks to cores, it offers lower runtime overheads and provides consistently good empirical performance at high utilizations [6].

Furthermore, taking the shared cache interference into account, partitioned approaches can achieve better schedulability than global scheduling. We provide a simple example to illustrate this. Consider three tasks τ_1, τ_2 and τ_3 with the same period and relative deadline of 7, the WCETs of τ_1, τ_2 and τ_3 are 3, 3 and 2, respectively. The execution platform is a processor with 2 cores including a last-level shared cache. If τ_1 and τ_2 run concurrently, we assume that the maximum cache interference exhibited by τ_1 and τ_2 is 3. We also assume that τ_3 has no cache interference with τ_1 and τ_2.

It is impossible to conclude that this taskset is schedulable under global scheduling. Figure 1 shows a case where τ_3 misses its deadline. At time $t = 0$, tasks τ_1 and τ_2 are scheduled to execute on the two cores. In the figure, the black area of a cumulative length of 3 denotes the $WCET$, and the hatched area of a cumulative length of 3 represents the extra execution time due to the cache interference. At $t = 6$, τ_1 and τ_2 both finish their executions, after which τ_3 starts its execution. At $t = 7$, τ_3 misses its deadline. Similarly, consider another case: at $t = 0$, τ_3 and τ_1 (or τ_2) are scheduled, at $t = 2$, τ_3 finishes and τ_2 (or τ_1) starts its execution. Since cache interference is counted per job [31], in the worst case, the cache interference exhibited by τ_2 (or τ_1) can still be 3 even though the duration of co-running τ_2 (or τ_1) and τ_1 (or τ_2) is less than in the previous case. Due to the cache interference, τ_2 (or τ_1) could finish its execution at $t = 8$, leading to a deadline miss for τ_2 (or τ_1).

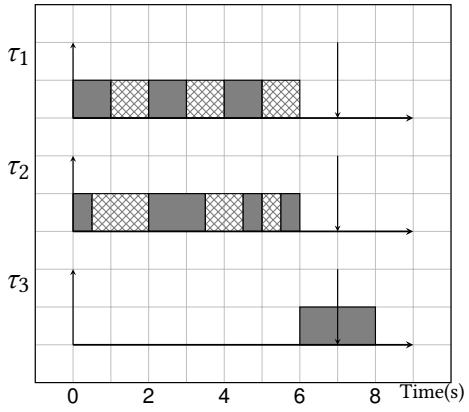


Figure 1. Case where τ_3 misses its deadline if τ_1, τ_2 and τ_3 are scheduled globally.

However, the taskset is schedulable under the partitioned scheduling. Consider, e.g., the partitioning scheme in which τ_1 and τ_2 are assigned to core 1, and task τ_3 is assigned to core 2. Since τ_1 and τ_2 are assigned to the same core, they cannot run simultaneously. As no cache interference can occur during task execution, it can be verified that every task meets its deadline.

Contributions. Motivated by the above example, in this work, we propose a novel cache interference-aware task partitioning algorithm, called CITTA. To the best of our knowledge, this is the first work on partitioned scheduling for real-time multi-core systems, accounting for shared cache interference. An integer programming formulation is constructed to calculate the upper bound on cache interference exhibited by a task, which is required by CITTA. We conduct schedulability analysis of CITTA and formally prove its correctness. A set of experiments is performed to evaluate the schedulability performance of CITTA against global EDF scheduling over randomly generated tasksets. Our empirical evaluations show that CITTA outperforms global EDF scheduling in terms of tasksets deemed schedulable.

The rest of the paper is organized as follows. Section 2 gives an overview of related work. The system model and some other prerequisites for this paper are described in Section 3. Section 4 describes the proposed CITTA, where we also detail the computation of the inter-core cache interference and schedulability analysis of CITTA. Section 5 presents the experimental results, after which Section 6 concludes the paper.

2 Related work

WCET estimation. For hard real-time systems, it is essential to obtain each real-time task's WCET, which provides the basis for the schedulability analysis. WCET analysis has been actively investigated in the last two decades, of which an excellent overview can be found in [30]. There are well-developed techniques to estimate a real-time tasks' WCET for single processor systems. Unfortunately, the existing techniques for single processor platforms are not applicable to multi-core systems with shared caches. Only a few methods have been developed to estimate task WCETs for multi-core systems with shared caches [17, 23, 36]. In almost all those works, due to the assumption that cache interferences can occur at any program point, WCET analysis will be extremely pessimistic, especially when the system contains many cores and tasks. An overestimated WCET is not useful as it degrades system schedulability.

Shared cache interference. Since shared caches make it difficult to accurately estimate the WCET of tasks, many researchers have recognized and studied the problem of cache interference in order to use shared caches in a predictable manner. Cache partitioning is a successful and widely-used approach to address contention for shared caches in (real-time) multi-core applications. There are two cache partitioning methods: software-based and hardware-based techniques [15]. The most common software-based cache partitioning technique is page coloring [24, 29]. By exploiting the virtual-to-physical page address translations present in virtual memory systems at OS-level, page addresses are mapped

to pre-defined cache regions to avoid the overlap of cache spaces. Hardware-based cache partitioning is achieved using a cache locking mechanism [9, 26, 28], which prevents cache lines from being evicted during program execution. The main drawback of cache locking is that it requires additional hardware support that is not available in many commercial processors for embedded systems.

A few works address schedulability analysis for multi-core systems with shared caches [16, 34], but these works use cache space isolation techniques to avoid cache contention for hard real-time tasks. In this work, we do not deploy any cache partitioning techniques to mitigate the inter-core cache interference. Instead, we address the problem of task partitioning in the presence of shared cache interference.

Real-time Scheduling. To schedule real-time tasks on multi-core platforms, different paradigms have been widely studied: partitioned [4, 13, 35], global [3, 7, 22], and semi-partitioned scheduling [8, 10, 20]. A comprehensive survey of real-time scheduling for multiprocessor systems can be found in [12]. Most multi-core scheduling approaches assume that the WCETs are estimated in an offline and isolated manner and that WCET values are fixed.

Real-time scheduling for multi-core systems using cache partitioning techniques is done via two steps: it first captures the relationship between the task's WCET and cache allocation by analysis or measurement as the WCET of a task depends on the number of cache partitions assigned to that task, and then develops a strategy that determines the number of cache partitions assigned to each task in the system, so that the task system is schedulable. Existing approaches typically adopt Mixed Integer Programming to find the optimal cache assignment. However, these methods incur a very high execution time complexity, and are therefore too inefficient to be practical [33].

Different from the above approaches based on cache partitioning techniques, we address the problem of task partitioning in the presence of shared cache interference. Our approach neither requires operating system modifications for page coloring nor hardware features for cache locking, which are not supported by most existing embedded processors.

The most relevant to our work is [31, 32], which also addresses schedulability analysis for multi-core systems with shared caches. However, the work of [31, 32] only considers global scheduling. In this paper, we consider another scheduling paradigm, namely partitioned scheduling, and propose CITTA, a cache interference-aware task partitioning algorithm. Our empirical evaluations show that CITTA outperforms global EDF scheduling in terms of task sets deemed schedulable.

3 System Model and Prerequisites

3.1 System Model

Task Model. A taskset τ comprises n periodic or sporadic real-time tasks $\tau_1, \tau_2, ... \tau_n$. Each task $\tau_k = (C_k, D_k, T_k) \in \tau$ is characterized by a worst-case computation time C_k, a period or minimum inter-arrival time T_k, and a relative deadline D_k. All tasks are considered to be deadline constrained, i.e. the task relative deadline is less or equal to the task period: $D_k \leq T_k$. We further assume that all those tasks are independent, i.e. they have no shared variables, no precedence constraints, and so on.

A task τ_k is a sequence of jobs J_k^j, where j is the job index. We denote the arrival time, starting time, finishing time and absolute deadline of a job j as r_k^j, s_k^j, f_k^j and d_k^j, respectively. Note that the goal of a real-time scheduling algorithm is to guarantee that each job will complete before its absolute deadline: $f_k^j \leq d_k^j = r_k^j + D_k$.

As explained, it is difficult to accurately estimate C_k considering cache interference of other tasks executing concurrently. It should be pointed out that C_k in this paper refers to the WCET of task k, assuming task k is the only task executing on the multi-core processor platform, i.e. any cache interference delays are not included in C_k.

Since time measurement cannot be more precise than one tick of the system clock, all timing parameters and variables in this paper are assumed to be non-negative integer values.

Our system architecture consists of a multi-core processor with m identical cores onto which the individual tasks are scheduled.

In multi-core processors, Caches are organized as a hierarchy of multiple cache levels to address the trade-off between cache latency and hit rate. The lower level caches, for example L1, are private while the last-level caches (*LLC*) are shared among all cores. The caches are assumed to be non-inclusive and direct-mapped.

Partitioned Non-preemtive Schedulers. In this paper, we focus on non-preemptive partitioned scheduling. Once a task instance starts execution, any preemption during the execution is not allowed, so it must run to completion. So we do not have to consider intra-core cache interference. If not explicitly stated, cache interference will therefore refer to inter-core cache interference in the following discussion.

Since partitioning tasks among a multi-core processor reduces the multi-core processor scheduling problem to a series of single-core scheduling problems (one for each core), the optimality without idle inserted time [14, 19] of non-preemptive EDF (EDF_{np}) makes it a reasonable algorithm to use as the run-time scheduler on each core. Therefore, we make the assumption that each core, and the tasks assigned to it by the partitioning algorithm, are scheduled at run time according to an EDF_{np} scheduler.

EDF_{np} assigns a priority to a job according to the absolute deadline of that job. A job with an earlier absolute deadline

has higher priority than others with a later absolute deadline. EDF_{np} scheduling is work-conserving: using EDF_{np}, there are no idle cores when a ready task is waiting for execution.

3.2 The Demand-Bound Function

A successful approach to analyzing the schedulability of real-time tasks is to use a demand bound function [5]. The demand bound function $DBF(\tau_i, t)$ is the largest possible cumulative execution demand of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within any time interval of length t. Let t_0 be the starting time of a time interval of length t, the cumulative execution demand of τ_i's jobs over $[t_0, t_0 + t]$ is maximized if one job arrives at t_0 and subsequent jobs arrive as soon as permitted i.e., at instants $t_0 + T_i, t_0 + 2T_i, t_0 + 3T_i$,... Therefore, $DBF(\tau_i, t)$ can be computed by Equation (0.1),

$$DBF(\tau_i, t) = max(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1) \times C_i). \quad (0.1)$$

[1] proposed a technique for approximating the $DBF(\tau_i, t)$. The approximated demand bound function $DBF^*(\tau_i, t)$ is given by the following equation:

$$DBF^*(\tau_i, t) = \begin{cases} 0 & t < D_i \\ C_i + U_i \times (t - D_i) & otherwise \end{cases} \quad (0.2)$$

where $U_i = \frac{C_i}{T_i}$.

Observe that the following inequality holds for all τ_i and all $0 \leq t$:

$$DBF^*(\tau_i, t) \geq DBF(\tau_i, t) \quad (0.3)$$

3.3 Uniprocessor Schedulability

The schedulabity analysis of uniprocessor scheduling is well studied. [2] presented a necessary and sufficient condition for the feasibility test of a sporadic task system τ scheduled by EDF_{np} on a uniprocessor platform.

Theorem 1. *A taskset τ is schedulable under EDF_{np} on a uniprocessor platform if and only if*

$$\forall t, \sum_{i=1}^{n} DBF(\tau_i, t) \leq t \quad (1.1)$$

and for all $\tau_j \in \tau$:

$$\forall t : C_j \leq t \leq D_j : C_j + \sum_{i=1; i \neq j}^{n} DBF(\tau_i, t) \leq t. \quad (1.2)$$

Note that the computation of $DBF(\tau_i, t)$ and $DBF^*(\tau_i, t)$ by Equation (0.1) and (0.2) and the two schedulability test conditions (1.1) and (1.2) do not account for shared cache interference. We will extend the computation of $DBF(\tau_i, t)$ and $DBF^*(\tau_i, t)$ and the two schedulability conditions to the cases where shared cache interference is considered.

3.4 Cache Interference

The WCET of a task can be obtained by performing a Cache Access Classification (CAC) and Cache Hit/Miss Classification (CHMC) analysis for each memory access at the private caches and the shared LLC cache separately [30]. The CAC categorizes the accesses to a certain cache level as Always (A), Uncertain (U) or Never (N). CHMC classifies the reference to a memory block as Always Hit (AH), Always Miss (AM) or Uncertain (U).

As an LLC is shared by multiple cores, it allows running tasks to compete among each other for shared cache space. As a consequence, the tasks replace blocks that belong to other tasks, causing shared cache interference. Let τ_k be the interfered and τ_i be the interfering task. We use $I^c_{i,k}$ to represent the upper bound on the shared cache interference imposed on τ_k by only one job execution of τ_i.

$I^c_{i,k}$ can be calculated, as indicated by Lemma 4 and its proof in [31], using the concept of Hit Block (HB), i.e. a memory block whose access is classified as AH at the shared cache and Conflicting Block (CB), i.e. memory block whose access is classified as A or U at the shared cache. By calculating the number of accesses to each τ_k's HB and the accesses to each τ_i's CB, $I^c_{i,k}$ can be derived by bounding the conflicting accesses to each shared cache set between τ_k and τ_i. In the following discussion, we assume $I^c_{i,k}$ is known.

4 Cache interference aware task partitioning : CITTA

Given a taskset τ comprised of n periodic or sporadic tasks and a processing platform π with m identical cores $\pi = \{\pi_1, \pi_2, ..., \pi_m\}$, a partitioning algorithm decides how to assign tasks to cores to avoid task deadline misses. The problem of assigning a set of tasks to a set of cores is analogous to the bin-packing problem. In this case, the tasks are the objects to pack and the bins are cores. The bin-packing problem is known to be NP-hard in the strong sense. Thus, searching for an optimal task assignment is not practical.

[25] and [13] studied several bin-packing heuristics for the preemptive and non-preemptive task model. Typically, each of the bin-packing heuristics follows the following pattern: tasks of the task system are first sorted by some criterion, after which the tasks are assigned in order to a core that satisfies a sufficient condition.

Let $\tau(\pi_x)$ denote the set of tasks assigned to processor core π_x where $1 \leq x \leq m$. $\tau_i \in \tau(\pi_x)$ means τ_i is assigned to core π_x. If taskset τ can be scheduled by a partitioned algorithm, the outcome of running a partitioning algorithm is a task partition such that:

- All tasks are assigned to processor cores:

$$\cup_{1 \leq x \leq m} \tau(\pi_x) = \tau$$

- Each task is assigned to only one core:

$$\forall y \neq x, 1 \leq y \leq m, 1 \leq x \leq m, \quad \tau(\pi_y) \cap \tau(\pi_x) = \emptyset$$

In Section 4.1, we describe our cache interference aware task partitioning : CITTA. Section 4.2 derives the calculation of the upper bound on the shared cache interference. Section 4.3 conducts the schedulability analysis for CITTA.

Before describing CITTA, we first extend the *DBF* to account for shared cache interference. Due to the extra execution delay caused by shared cache interference, a task τ_i may execute longer than C_i. Given a task partitioning scheme, one can compute the upper bound on cache interference exhibited by task τ_i, denoted as $\bar{I}_i^c$. We will show the method to compute this $\bar{I}_i^c$ later. In multiprogrammed environment, the actual execution time including cache interference of τ_i can be bounded by $C_i + \bar{I}_i^c$. We denote $DBF^c(\tau_i, t)$ as the demand bound function which accounts for cache interference. $DBF^c(\tau_i, t)$ can be computed by extending Equation (0.1):

$$DBF^c(\tau_i, t) = max(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1) \times (C_i + \bar{I}_i^c)). \quad (1.3)$$

Similarly, the approximated demand bound function $DBF^{c*}(\tau_i, t)$ is given by the following equation by extending Equation (0.2):

$$DBF^{c*}(\tau_i, t) = \begin{cases} 0 & t < D_i \\ C_i + \bar{I}_i^c + U_i^c \times (t - D_i) & otherwise \end{cases} \quad (1.4)$$

where $U_i^c = \frac{C_i + \bar{I}_i^c}{T_i}$.

It can also be observed that:

$$DBF^{c*}(\tau_i, t) \geq DBF^c(\tau_i, t) \quad (1.5)$$

4.1 The Task Partitioning Algorithm: CITTA

We now propose CITTA, a task partitioning algorithm taking shared cache interference into account.

We assume the tasks are sorted in non-decreasing order by means of a certain criterion. For example, if a task's relative deadline is chosen as criterion, then $D_i \leq D_{i+1}$ for $1 \leq i \leq n$. More criteria for sorting the tasks will be discussed in Section 5.

CITTA performs the following steps:

step 1: for each task $\tau_i \in \tau$:

1. **Attempt** to assign τ_i to π_x,
2. Calculate the upper bound on cache interference $\bar{I}_k^c$ for $\tau_k \in \tau(\pi_x) \cup \{\tau_i\}$, i.e. tasks that are already assigned to π_x and τ_i, assuming τ_i is assigned to π_x. We will show the calculation procedure in the next subsection.
3. Check if the following condition holds for each $\tau_k \in \tau(\pi_x) \cup \{\tau_i\}$

$$D_k \geq \sum_{\substack{\tau_j \in \tau(\pi_x) \cup \{\tau_i\} \\ D_j \leq D_k}} DBF^{c*}(\tau_j, D_k) + \max_{\substack{\tau_j \in \tau(\pi_x) \cup \{\tau_i\} \\ D_j > D_k}} C_j + \bar{I}_j^c. \quad (1.6)$$

   a. If no τ_k violates condition (1.6), the attempt is **admitted** and τ_i is added to $\tau(\pi_x)$.
   b. If condition (1.6) is violated by at least one τ_k, the attempt is **rejected**. We attempt to assign τ_i to the next core π_{x+1} and repeat steps (2) and (3). If no core can be assigned to τ_i, then τ_i is added to the temporarily non-allocable taskset, denoted as τ^{tna}.

step 2: after performing step 1, the resulting τ^{tna} is either an empty set or non-empty.

(a) If $\tau^{tna} = \emptyset$, which means all tasks have been allocated to cores, CITTA returns *Success*,

(b) Otherwise, we perform step 1 to each $\tau_t \in \tau^{tna}$. τ_t is removed from τ^{tna} if it can be assigned to a core. We repeatedly perform step 1 to $\tau_t \in \tau^{tna}$ until τ^{tna} becomes empty or no more tasks in τ^{tna} could be allocated to cores. If $\tau^{tna} = \emptyset$ at the end, CITTA returns *Success*, otherwise CITTA returns *Fail*: it is unable to determine if scheduling τ is feasible on the multi-core platform.

We briefly explain the rationale behind condition (1.6). Given a task τ_k, the execution demand of tasks (including τ_k) with a relative deadline no larger than D_k is calculated by the first part (left-hand side) of the sum in condition (1.6). Since we consider a non-preemptive task system, the second part of the sum accounts for the blocking time due to the execution of a task with a larger relative deadline than τ_k at the time a job of τ_k arrives. If the sum of the execution demand and the blocking time is smaller than D_k, the task τ_k will not miss its deadline. We will prove this in Section 4.3.

A more formal version of the task partitioning algorithm CITTA is given by Pseudocode 1. The input to procedure *CITTA* is the taskset τ to be partitioned and the execution platform π consisting of m cores. *CITTA* repeatedly invokes the procedure *TaskPartition*, illustrated by Pseudocode 2, to perform step 1 of the CITTA algorithm. The input to *TaskPartition* is the temporarily non-allocable taskset τ^{tna}, π, and existing task assignment $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), ..., \tau(\pi_m))$. τ^{tna} is initialized as τ. Every time when *TaskPartition* finishes, some tasks in the taskset τ^{tna} can be assigned to cores, and thus τ^{tna} and $\tau(\pi)$ are updated.

Pseudocode 1: *CITTA*(τ, π)

```
1:  sort τ in non-decreasing order by a selected criterion
2:  τ^tna ← τ, taskAssigned ← true, τ(π_1), τ(π_2), ..., τ(π_m)
    ← ∅
3:  τ(π) = (τ(π_1), τ(π_2), ..., τ(π_m))
4:  while τ^tna ≠ ∅ and taskAssigned == true do
5:      τ^tna, taskAssigned, τ(π)=TaskPartition(τ^tna, π, τ(π))
6:  end while
7:  if τ^tna == ∅ then
8:      return Success
9:  else
10:     return Failed
11: end if
```

Lines 5 – 7 in the procedure of *TaskPartition* perform step 1.(2) of CITTA to compute the upper bound on cache interference for tasks. When CITTA attempts to assign τ_i to π_x, the upper bound on cache interference caused by $\tau_k \in \tau(\pi_x)$,

Pseudocode 2: *TaskPartition*(τ, π, $\tau(\pi)$)

```
taskAssigned ← false, τ^tna ← ∅
for all τ_i ∈ τ do
    assignTo ← NULL, coreSuccess ← true
    for all π_x ∈ π do
        for all τ_k ∈ τ(π_x) ∪ {τ_i} do
            calculate Ī_k^c
        end for
        for all τ_k ∈ τ(π_x) ∪ {τ_i} do
            if condition (1.6) violates for τ_k then
                coreSuccess ← false
                break;
            end if
        end for
        if coreSuccess then
            τ(π_x) ← τ(π_x) ∪ {τ_i}
            assignTo ← π_x, taskAssigned ← true
            break;
        end if
    end for
    if assignTo == NULL then
        τ^tna ← τ^tna ∪ {τ_i}
    end if
end for
return τ^tna, taskAssigned, τ(π)
```

i.e. tasks that are already assigned to π_x, is recomputed. This is because a tighter bound can be possibly obtained by the recalculation, as will be shown soon. Considering τ_i is more likely to be assigned to π_x if the upper bound on the cache interference caused by $\tau_k \in \tau(\pi_x)$ is smaller, the recalculation makes CITTA less pessimistic.

4.2 Calculation of The Upper Bound on Cache Interference: $\bar{I}_k^c$

The CITTA algorithm requires to calculate the upper bound on cache interference before it assigns a new task to a core. We now describe such a procedure for the calculation of $\bar{I}_k^c$.

[31] presented an approach to calculating the upper bound on cache interference for tasks that are globally scheduled. By extending the approach in [31], we compute the upper bound on cache interference for partitioned scheduling. This is done by two steps. First, given the existing task assignment represented by $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), ..., \tau(\pi_m)$ and τ^{na} as the taskset consisting of the tasks that have not been assigned, we construct an integer programming (*IP*) formulation to calculate the upper bound on the cache interference exhibited by a task within an execution window. Then, we use an iterative algorithm to obtain the upper bound on cache interference a task may exhibit during its job executions.

4.2.1 IP formulation

In the following discussion, we compute the upper bound on cache interference exhibited by τ_k, assuming τ_i is the interfering task and τ_k is assigned to π_x.

The Execution Window (EW) of the j-th job of τ_k (J_k^j) is defined as the time interval $[s_k^j, f_k^j]$ from the staring time to the finishing time of J_k^j. We use C'_k as the length of the *EW* because of the iterative computation which will be described later on.

The objective function of the *IP* formulation is to maximize the the total cache interference exhibited by task τ_k. If $N_{i,k}$ jobs of τ_i are executing concurrently with τ_k, the cache interference that τ_i causes on τ_k is bounded by $N_{i,k} \cdot I_{i,k}^c$. The total cache interference for one job execution of τ_k is bounded by the sum of the contributions of all tasks τ_i in the taskset τ. So the objective function is:

$$\max \sum N_{i,k} \cdot I_{i,k}^c. \tag{1.7}$$

To get a bounded solution, we analyze the constraints on $N_{i,k}$.

If tasks τ_i and τ_k are assigned to the same core π_x, at each time instance, at most one task of τ_i and τ_k executes on core π_x. No jobs from τ_i could interfere with τ_k. Therefore, we have the following:

$$\forall \tau_i \in \tau(\pi_x), N_{i,k} = 0. \tag{1.8}$$

$N_{i,k}$ reaches its minimal value when a job of τ_i starts to execute as soon as it is released and the execution finishes just before the start of the *EW*. Taking the smallest execution time of τ_i, C_i^{min}, as 0, we have the following constraint:

$$\forall \tau_i \notin \tau(\pi_x), \left\lfloor \frac{\max(0, C'_k - T_i)}{T_i} \right\rfloor + \xi_i \leq N_{i,k} \tag{1.9}$$

where $\xi_i = \begin{cases} 1 & (C'_k \mod T_i) - D_i > 0 \\ 0 & otherwise \end{cases}$.

The term ξ_i indicates whether or not the last job of τ_i released within the *EW* interferes with τ_k. The maximum value of $N_{i,k}$ is taken when the first interfering job of τ_i finishes just after the start of the *EW* and the last interfering job of τ_i starts to execute at the time when it is released. Thus, we have the second constraint on $N_{i,k}$:

$$\forall \tau_i \notin \tau(\pi_x),\ N_{i,k} \leq 1 + \left\lceil \frac{\max(0, C'_k - T_i + D_i)}{T_i} \right\rceil. \tag{1.10}$$

If $N_{i,k} > 2$, the first and last interfering jobs of τ_i may occupy almost 0 computation capacity in the *EW*. Let J_i^j be a job among the remaining $N_{i,k} - 2$ interfering jobs of τ_i between the first and the last ones. Both release time r_i^j and deadline d_i^j of J_i^j are within the EW of τ_k.

If τ_i is (or will be) successfully assigned to core π_y , at least C_i computation capacity of the processing core is reserved for the execution of J_i^j during $[r_i^j, d_i^j]$. The total execution of interfering tasks τ_i on each processor y (with $y \neq x$) cannot

exceed C'_k. Since we do not know the core assignment for tasks in τ^{na}, those tasks are allowed to execute on any core. Thus, we have the following inequality (1.11),

$$\forall y \neq x, \sum_{\tau_i \in \tau(\pi_y) \cup \tau^{na}} \max(0, N_{i,k} - 2)C_i \leq C'_k. \quad (1.11)$$

The objective function (1.7) together with constraints on $N_{i,k}$ i.e. inequalities (1.8), (1.9), (1.10) and (1.11) form our *IP* problem. As task parameters such as C_i, D_i, T_i are known, the input of the *IP* formulation is the length of EW: C'_k, existing task assignment: $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), ..., \tau(\pi_m)$, and remaining tasks that need to be assigned: τ^{na}. Thus, we use $IP(C'_k, \tau(\pi), \tau^{na})$ to denote the *IP* problem and use $I^c(C'_k, \tau(\pi), \tau^{na})$ to denote the optimal solution.

When CITTA attempts to assign a task τ_i to a core π_x, the upper bound on cache interference caused by $\tau_k \in \tau(\pi_x)$, i.e. tasks that are already assigned to π_x, is recomputed. We now show that a tighter upper bound for task τ_k can be possibly obtained by the re-computation.

Given a task τ_k and an execution window of length C'_k, let us suppose the *IP* formulation in the previous computation of cache interference is $IP(C'_k, \tau_p(\pi), \tau_p^{na})$, and the *IP* formulation for the re-computation is $IP(C'_k, \tau_q(\pi), \tau_q^{na})$.

Between the two computations for the same task τ_k, CITTA may assign some tasks to cores. If a task τ_i is assigned to a core π_x, τ_i is removed from τ_p^{na} and is added to $\tau_q(\pi_x)$. Obviously, we have $\tau_q^{na} \subseteq \tau_p^{na}$ and $\forall 1 \leq x \leq m, \tau_p(\pi_x) \subseteq \tau_q(\pi_x)$.

Lemma 1. *Given τ_k and C'_k,*

$$I^c(C'_k, \tau_q(\pi), \tau_q^{na}) \leq I^c(C'_k, \tau_p(\pi), \tau_p^{na}).$$

Proof Sketch: Due to space considerations, we will only show the proof sketch.

From condition 1.6, one can prove the following: if $\tau_i \in \tau(\pi_x)$ and $\tau_k \in \tau(\pi_x)$, then $C_k + \bar{I}_k^c \leq D_i$.

By the above statement and the constraints of the IP problem, we can prove that any solution of $IP(C'_k, \tau_q(\pi), \tau_q^{na})$ is also feasible for $IP(C'_k, \tau_p(\pi), \tau_p^{na})$. Thus,

$$I^c(C'_k, \tau_q(\pi), \tau_q^{na}) \leq I^c(C'_k, \tau_p(\pi), \tau_p^{na}).$$

Lemma 1 is the reason CITTA forces the recalculation of upper bound on cache interference caused by tasks that are already assigned to cores by CITTA.

4.2.2 Iterative Computation

Due to the presence of cache interference, a job may execute longer than C_k on a multi-core platform with shared caches. However, a larger execution time may introduce more cache interference.

We give a sufficient condition for a certain value that can be used as an upper bound on cache interference exhibited by τ_k, denoted by $\bar{I}_k^c$.

Lemma 2. *Given $\tau(\pi)$ and τ^{na}, if $\exists\, C_k^* \geq C_k$ such that $C_k^* = C_k + I^c(C_k^*, \tau(\pi), \tau^{na})$, then $\bar{I}_k^c = I^c(C_k^*, \tau(\pi), \tau^{na})$.*

The equation can be solved by means of fixed point iteration: the iteration starts with an initial value for the length of *EW* and upper bound on cache interference, i.e. $C'_k = C_k$ and $I^c(C'_k) = 0$. By solving the IP, we compute a new upper bound of the cache interference $I^c(C'_k, \tau(\pi), \tau^{na})$ and a new corresponding length of *EW*, $C'_k = C_k + I^c(C'_k, \tau(\pi), \tau^{na})$. The iterative computation for τ_k stops either if no update on $I^c(C'_k, \tau(\pi), \tau^{na})$ is possible anymore or if the computed $I^c(C'_k, \tau(\pi), \tau^{na})$ is large enough to make τ_k unschedulable i.e. $I^c(C'_k, \tau(\pi), \tau^{na}) + C'_k > D_k$.

Computational complexity: The original *IP* can be easily transformed to an Integer Linear Programming (*ILP*) problem by introducing a new integer variable $y_{i,k}$ for each $N_{i,k}$ with two additional constraints: $y_{i,k} \geq 0$ and $y_{i,k} \geq N_{i,k} - 2$. Inequality (1.11) can be replaced by $\sum_{\tau_i \in \tau(\pi_y) \cup \tau^{na}} y_{i,k} C_i \leq C'_k$. In the transformed *ILP* problem, we have totally $2n$ variables and $4n + m - 1$ constraints. The complexity of the *IP* is the same as the complexity of solving the transformed *ILP* problem, which is $O((4n + m)64^n \ln 4n + m)$ [11].

Let n represent the number of tasks in the taskset. For τ_k, let I_k^{min} be the smallest difference between cache interference caused by one job of τ_i and τ_j, i.e. $I_k^{min} = \min_{i,j}(I_{i,k}^c - I_{j,k}^c)$, the iterative algorithm takes at most $\gamma = \max_k \frac{(D_k - C_k)}{I_k^{min}}$ iterations to terminate since C'_k either stays the same or increases at least with I_k^{min} in each iteration. Thus, the complexity to compute the upper bound on cache interference exhibited by each task is $O(\gamma(4n^2 + mn)64^n ln4n + m)$. In *TaskPartition*, at most n tasks in τ are checked for at most m cores, thus, the complexity of *TaskPartition* is $O(\gamma(4n^2m + nm^2)64^n ln4n + m)$. Since the while loop in *CITTA* executes at most n times, the complexity of CITTA is $O(\gamma(4n^3m + m^2n^2)64^n ln4n + m)$.

4.3 Schedulability Analysis

4.3.1 Uniprocessor feasibility

Task partitioning reduces the problem of multi-core processor scheduling into a set of single-core processor scheduling problems (one for each core). Following Theorem 1, we first propose a schedulability condition, as stated in Theorem 2, for uniprocessor scheduling, taking shared cache interference into consideration. Note that the condition in Theorem 2 is sufficient and not necessary as $\bar{I}_j^c$ is the calculated upper bound on the shared interference exhibited by τ_j, the actual cache interference can be smaller than $\bar{I}_j^c$.

Theorem 2. *A taskset $\tau(\pi_x)$ is schedulable under EDF_{np} on a uniprocessor platform if*

$$\forall t, \sum_{\tau_i \in \tau(\pi_x)} DBF^c(\tau_i, t) \leq t \quad (2.1)$$

and for all $\tau_j \in \tau(\pi_x)$:

$$\forall t : C_j + \bar{I}_j^c \leq t \leq D_j : C_j + \bar{I}_j^c + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ i \neq j}} DBF^c(\tau_i, t) \leq t. \quad (2.2)$$

4.3.2 Schedulability analysis of CITTA

We first derive one property that must be satisfied for tasks assigned to the same core by CITTA. This is useful for the proof of the feasibility analysis conducted later for CITTA.

Lemma 3. *If tasks are assigned to cores by CITTA,*

$$\forall \pi_x \in \pi, \sum_{\tau_i \in \tau(\pi_x)} U_i^c \leq 1. \tag{2.3}$$

Proof. Let τ_u be the task with the largest relative deadline among tasks in $\tau(\pi_x)$, so, $D_u = max\{D_i | \tau_i \in \tau(\pi_x)\}$. Obviously,

$$\tau_i \in \tau(\pi_x) \Longrightarrow D_i \leq D_u.$$

Since τ_u satisfies Inequality (1.6), we have

$$D_u \geq \sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, D_u). \tag{2.4}$$

From Equation (1.4), $DBF^{c*}(\tau_i, D_u)$ is computed by:

$$DBF^{c*}(\tau_i, D_u) = U_i^c \times (D_u - D_i + T_i) \geq U_i^c \times D_u.$$

Replacing $DBF^{c*}(\tau_i, D_u)$ in Inequality (2.4),

$$D_u \geq \sum_{\tau_i \in \tau(\pi_x)} U_i^c \times D_u \Longrightarrow \sum_{\tau_i \in \tau(\pi_x)} U_i^c \leq 1.$$

This is Inequality (2.3). □

On each core $\pi_x \in \pi$, tasks in $\tau(\pi_x)$ are scheduled under EDF_{np}. The next lemma shows the feasibility of $\tau(\pi_x)$.

Lemma 4. *If the tasks are assigned to cores by CITTA, $\forall \pi_x \in \pi$, $\tau(\pi_x)$ is feasible on core π_x by EDF_{np}.*

Proof. For the sake of contradiction, assume that each task in $\tau(\pi_x)$ satisfies condition (1.6), but that a task's deadline is missed when scheduling the tasks in $\pi(\tau_k)$ on core π_x. Let t_f be the time that a task misses a deadline on core π_x.

By Theorem 2, either

$$\sum_{\tau_i \in \tau(\pi_x)} DBF^c(\tau_i, t_f) > t_f, \tag{2.5}$$

or $\exists \tau_p, \tau_p \in \tau(\pi_x)$ and $\exists t_f, C_p + \bar{I}_p^c \leq t_f \leq D_p$, such that

$$C_p + \bar{I}_p^c + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ i \neq p}} DBF^c(\tau_i, t_f) > t_f. \tag{2.6}$$

It will be shown that if either Inequality (2.5) or (2.6) holds, then a contradiction is reached.

We first prove the existence of $\tau_i \in \tau(\pi_x)$ that satisfies $D_i \leq t_f$. Assuming $\forall \tau_i \in \tau(\pi_x), D_i > t_f$, from Equation (1.4),

$$\sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f) = 0.$$

By the assumption, neither Inequality (2.5) nor (2.6) will hold. So the assumption is false.

Therefore, we can always find $\tau_i \in \tau(\pi_x)$ that satisfies $D_i \leq t_f$. Let τ_s be the task with the largest relative deadline, i.e. $D_s = max\{D_i | \tau_i \in \tau(\pi_x) \wedge D_i \leq t_f\}$

(A) we first prove that if Inequality (2.5) holds, it would lead to contradiction.

From Inequality (1.5) and (2.5),

$$\sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f) > t_f. \tag{2.7}$$

By the definition of $DBF^{c*}(\tau_i, t_f)$, we have

$$\sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i > D_s}} DBF^{c*}(\tau_i, t_f) = 0.$$

$$\begin{aligned}
&\sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f) \\
&= \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, t_f) + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i > D_s}} DBF^{c*}(\tau_i, t_f) \\
&= \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} C_i + \bar{I}_i^c + U_i^c \times (t_f - D_i) \\
&= \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} C_i + \bar{I}_i^c + U_i^c \times (t_f - D_s + D_s - D_i) \\
&= \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, D_s) + U_i^c \times (t_f - D_s).
\end{aligned} \tag{2.8}$$

τ_s satisfies condition (1.6):

$$D_s \geq \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, D_s).$$

From Equation (2.8) and Inequality (2.7), we have

$$D_s + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} U_i^c \times (t_f - D_s) > t_f \tag{2.9}$$

$$\Longrightarrow \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} U_i^c > 1 \Longrightarrow \sum_{\tau_i \in \tau(\pi_x)} U_i^c > 1.$$

This contradicts to Lemma 3.

(B) we now prove that if Inequality (2.6) holds, it would also lead to contradiction.

We know that $\exists \tau_s, \tau_p$ such that $D_s \leq t_f \leq D_p$. We consider two cases (B1): $D_s = D_p$ and (B2): $D_s < D_p$.

(B1) if $D_s = D_p$, then $t_f = D_p$

$$DBF^{c*}(\tau_p, t_f) = C_p + \bar{I}_p^c$$

From Inequality (2.6),

$$\sum_{\tau_i \in \tau(\pi_x)} DBF^c(\tau_i, t_f) > t_f.$$

This leads to contradiction as proved in case (A).

(B2) if $D_s < D_p$, we have

$$C_p + I_p^c \leq \max_{\substack{\tau_j \in \tau(\pi_x) \\ D_j > D_s}} C_j + \bar{I}_j^c,$$

and

$$\sum_{\substack{\tau_i \in \tau(\pi_x) \\ i \neq p}} DBF^c(\tau_i, t_f) \leq \sum_{\tau_i \in \tau(\pi_x)} DBF^c(\tau_i, t_f).$$

From Inequality (2.6), we have

$$\max_{\substack{\tau_j \in \tau(\pi_x) \\ D_j > D_s}} C_j + \bar{I}_j^c + \sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f) > t_f.$$

Replacing $\sum_{\tau_i \in \tau(\pi_x)} DBF^{c*}(\tau_i, t_f)$ in the above inequality using equation (2.8), we have

$$\max_{\substack{\tau_j \in \tau(\pi_x) \\ D_j > D_s}} C_j + \bar{I}_j^c + \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, D_s) + U_i^c \times (t_f - D_s) > t_f. \tag{2.10}$$

Since τ_s satisfies condition (1.6),

$$D_s \geq \sum_{\substack{\tau_i \in \tau(\pi_x) \\ D_i \leq D_s}} DBF^{c*}(\tau_i, D_s) + \max_{\substack{\tau_i \in \tau(\pi_x) \\ D_i > D_s}} C_i + \bar{I}_i^c. \tag{2.11}$$

From Inequality (2.10) and (2.11),

$$\sum_{\tau_i \in \tau(\pi_x)} U_i^c > 1.$$

This also contradicts to Lemma 3. □

The correctness of Algorithm CITTA follows, by application of Lemma 4:

Theorem 3. *If the task partitioning algorithm CITTA returns Success on taskset τ, then the resulting partitioning is schedulable by EDF_{np} on each core.*

5 Experiments

We asset the performance of CITTA and the proposed schedulability test in terms of acceptance ratio, that is, the number of tasksets that are deemed schedulable divided by the number of tasksets tested. CITTA is compared against Global EDF (GEDF), which is proposed in [32], the only, at least to the best of our records, work on real-time multiprocessor scheduling, taking the shared cache interference into account.

As mentioned in the beginning of Section 4.1, the CITTA algorithm first sorts tasks in non-decreasing order using some criterion and then assigns tasks to the processor cores according to Equations (1.6).

We consider the following five sorting criteria: the reciprocal of a task's WCET $\frac{1}{C_i}$, a task's period T_i, the reciprocal of a task's utilization $\frac{1}{U_i} = \frac{T_i}{C_i}$, a task's slack $S_i = T_i - C_i$ and *random* order.

5.1 Workloads Generation

We systematically generated synthetic workloads by varying i) the number of tasks n ($n = 10,\ 20$) in the taskset, ii) total task utilization U_{tot} (U_{tot} from 0.1 to $m - 0.1$ with steps of 0.2), iii) the cache interference factor IF ($IF = 0.2$ or 0.8), and iv) the probability of two tasks having cache interference on each other: P ($P = 0.1$ or 0.4). Given those four parameters, we have generated 20000 tasksets in each experiment.

We adopted the same policy, described in [31], to generate task parameters such as task period and utilization, and cache interference between two tasks.

In each experiment, we measure the number of tasksets that can be successfully partitioned by CITTA with different sorting criteria and the number of tasksets that can be scheduled by *GEDF*. The acceptance ratio is the number of schedulable tasksets divided by the total number of tasksets.

5.2 Results

We report the major trends characterizing the experimental results, illustrated in Figures 2 and 3. In the figures, CITTA-<*criterion*> represents a variant of CITTA using <*criterion*> for sorting tasks, GLB stands for the GEDF scheduler.

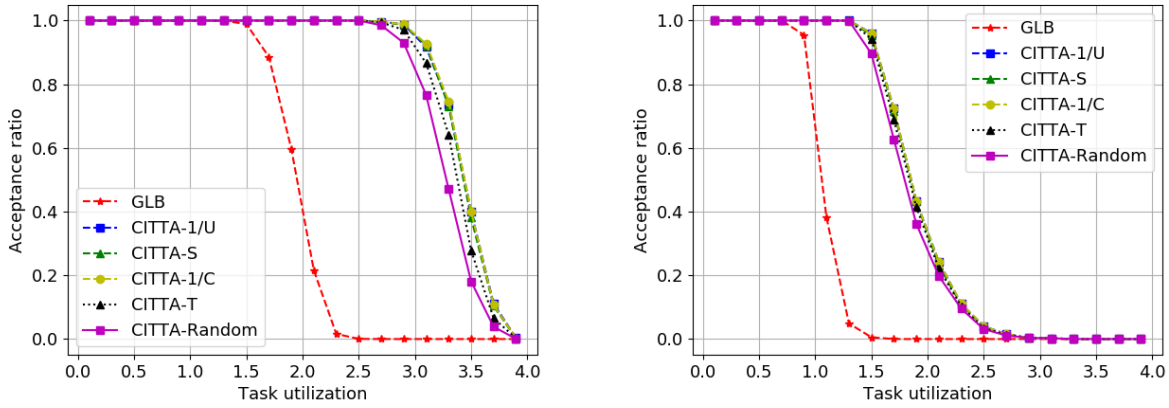
CITTA outperforms global EDF. Our results clearly show that CITTA outperforms global EDF in all the test cases. It is also evident that CITTA is highly effective for multi-core real-time systems, accounting for cache interference.

As shown in Figure 2(a), when $IF = 0.2, P = 0.1$, all the generated tasksets can be successfully partitioned by all variants of CITTA if $U_{tot} < 2.5$. while the global EDF achieves the full acceptance ratio when $U_{tot} < 1.5$. CITTA is able to partition tasksets with the highest tested total utilization, i.e. $U_{tot} = 3.9$. Global EDF can only schedule tasksets with a total utilization of up to $U_{tot} = 2.5$.

It is important to observe that the gap of acceptance ratio between all variants of CITTA and global scheduling is large when $U_{tot} \in [2, 3.5]$. Such a schedulability performance gap also exists for different degrees of cache interference and different numbers of tasks in the taskset, as shown in Figure 2(b), Figure 3(a) and Figure 3(b).

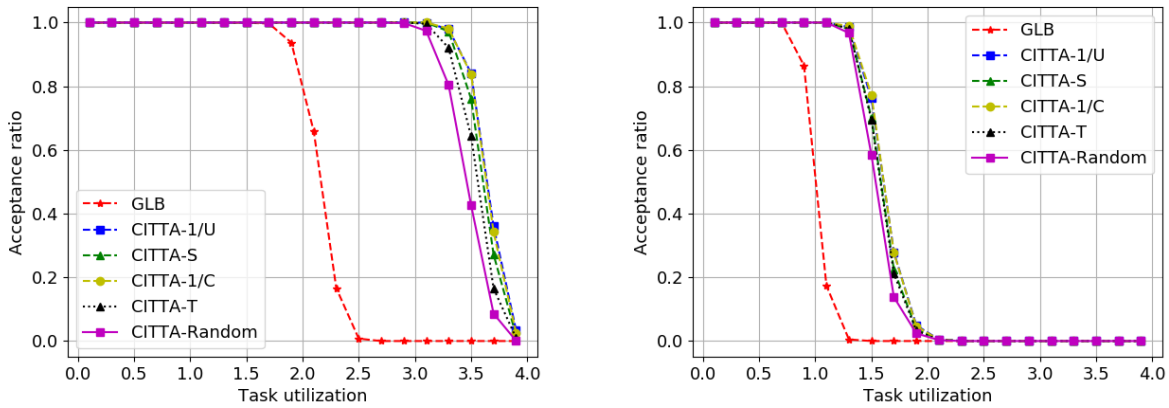
We have also compared the schedulability performance of CITTA and GEDF using heterogeneous task periods i.e. $T_i \in [100, 300]$ or $T_i \in [100, 500]$ (of which the results are omitted due to space limitations). In those tests, CITTA still outperforms GEDF.

Performance gap among different variants of CITTA is small. As is depicted in Figures 2(a) and 3(a), when the cache interference is small ($IF = 0.2, P = 0.1$), CITTA-*T* and CITTA-*random* performed worse than the CITTA-1/*C*, CITTA-*S* and CITTA-1/*U* when $U_{tot} > 3$. while as the degree of cache interference increases, the schedulability performance gap becomes smaller, as shown in Figure 2(b) and Figure 3(b). One reason could be that even though tasks are sorted by different criteria, all variants of CITTA force recalculation of the upper bound on cache interference to obtain an upper bound that is as small as possible. The cache interference obtained by all variants of CITTA thus is likely to be similar. Therefore, if cache interference dominates the schedulability result, the gap of schedulability performance among different variants of CITTA is small.



(a) IF=0.2, P=0.1. (b) IF=0.4, P=0.8.

Figure 2. Acceptance ratio with different IF and P when $m = 4$, $n = 10$.



(a) IF=0.2, P=0.1. (b) IF=0.4, P=0.8.

Figure 3. Acceptance ratio with different IF and P when $m = 4$, $n = 20$.

Cache interference degrades schedulability performance. Figure 2(a) and Figure 2(b) compare the acceptance ratio with different P and IF for tasksets consisting of 10 tasks. With the same U_{tot}, the acceptance ratio achieved by all variants of CITTA and global EDF decrease as P and IF increase. This is because a larger P and IF indicate more tasks in the taskset having larger cache interference with each other, which can potentially increase the upper bound on cache interference, eventually making the interfered tasks unschedulable. Similar observation can be made from Figure 3(a) and Figure 3(b) for tasksets consisting of 20 tasks.

5.3 Average Execution Time

We measured the execution time of CITTA with different taskset sizes. The executions are conducted on an Intel Xeon processor using only one core running at $2.4GHz$. On average, it takes 0.85 seconds to run CITTA for assignment of the taskset consisting of 10 tasks to a processor with 4 cores, while it takes 2.3 seconds for tasksets with 20 tasks.

6 Conclusions

Shared caches in multi-core processors introduce serious difficulties in providing guarantees on the real-time properties of embedded software. In this paper, we addressed the problem of task partitioning in the presence of cache interference. To achieve this, CITTA, a cache-interference aware task partitioning algorithm was proposed. An integer programming formulation was constructed to calculate the upper bound on cache interference exhibited by a task, which is required by CITTA. We conducted schedulability analysis of CITTA and formally proved the correctness of CITTA. A set of experiments was performed to evaluate the schedulability performance of CITTA against global EDF scheduling over randomly generated tasksets. Our empirical evaluations shows that CITTA outperforms global EDF scheduling in terms of tasksets deemed schedulable. As for future work, we plan to combine the task partitioning and cache partitioning approaches to design a new real-time scheduling algorithm that can achieve even better schedulability.

References

[1] K. Albers and F. Slomka. 2004. An event stream driven approximation for the analysis of real-time systems. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* 187–195.

[2] Sanjoy Baruah. 2005. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05).* 137–144.

[3] Sanjoy Baruah. 2007. Techniques for Multiprocessor Global Schedulability Analysis. In *RTSS'07 (RTSS '07).* IEEE Computer Society, Washington, DC, USA, 119–128.

[4] S. Baruah and N. Fisher. 2005. The partitioned multiprocessor scheduling of sporadic task systems. In *26th IEEE International Real-Time Systems Symposium (RTSS'05).* 9 pp.–329.

[5] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. 1990. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *In Proceedings of the 11th Real-Time Systems Symposium.* IEEE Computer Society Press, 182–190.

[6] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. 2010. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers. In *2010 31st IEEE Real-Time Systems Symposium.* 14–24. https://doi.org/10.1109/RTSS.2010.23

[7] M. Bertogna, M. Cirinei, and G. Lipari. 2009. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Transactions on Parallel and Distributed Systems* 20, 4 (April 2009), 553–566. https://doi.org/10.1109/TPDS.2008.129

[8] B. B. Brandenburg and M. GÃijl. 2016. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *2016 IEEE Real-Time Systems Symposium (RTSS).* 99–110.

[9] Marco Caccamo, Marco Cesati, Rodolfo Pellizzoni, Emiliano Betti, Roman Dudko, and Renato Mancuso. 2013. Real-time Cache Management Framework for Multi-core Architectures. In *RTAS' 13 (RTAS '13).* IEEE Computer Society, Washington, DC, USA, 45–54.

[10] Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. 2017. Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Marko Bertogna (Ed.), Vol. 76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:23.

[11] Kenneth L. Clarkson. 1995. Las Vegas algorithms for linear and integer programming when the dimension is small. *J. ACM* 42 (1995), 488–499.

[12] Robert I. Davis and Alan Burns. 2011. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.* 43, 4, Article 35 (Oct. 2011), 44 pages. https://doi.org/10.1145/1978802.1978814

[13] Nathan Fisher and Sanjoy Baruah. 2006. The partitioned multiprocessor scheduling of non-preemptive sporadic task systems. In *14th International conference on real-time and network systems.*

[14] Laurent George, Paul Muhlethaler, and Nicolas Rivierre. 1995. *Optimality and non-preemptive real-time scheduling revisited.* Research Report RR-2516. INRIA. Projet REFLECS.

[15] G. Gracioli and A. A. Fröhlich. 2013. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In *RTCSA' 03.* 72–81.

[16] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *7th ACM international conference on Embedded software.* ACM, 245–254.

[17] D. Hardy, T. Piquet, and I. Puaut. 2009. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *RTSS '09.* 68–77.

[18] D. Hardy and I. Puaut. 2008. WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches. In *RTSS'08.* 456–466.

[19] K. Jeffay, D. F. Stanat, and C. U. Martel. 1991. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings Twelfth Real-Time Systems Symposium.* 129–139.

[20] S. Kato and N. Yamasaki. 2009. Semi-partitioned Fixed-Priority Scheduling on Multiprocessors. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium.* 23–32.

[21] H. Kim, A. Kandhalu, and R. Rajkumar. 2013. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *ECRTS' 13.* 80–89.

[22] J. Lee, K. G. Shin, I. Shin, and A. Easwaran. 2015. Composition of Schedulability Analyses for Real-Time Multiprocessor Systems. *IEEE Trans. Comput.* 64, 4 (April 2015), 941–954. https://doi.org/10.1109/TC.2014.2308183

[23] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. 2009. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *2009 30th IEEE Real-Time Systems Symposium.* 57–67. https://doi.org/10.1109/RTSS.2009.32

[24] J. Liedtke, H. Hartig, and M. Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *RTAS' 97.* 213–224.

[25] José María López, José Luis Díaz, and Daniel F García. 2004. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems* 28, 1 (2004), 39–68.

[26] Mayank Shekhar, Abhik Sarkar, Harini Ramaprasad, and Frank Mueller. 2012. Semi-Partitioned Hard-Real-Time Scheduling Under Locked Cache Migration in Multicore Systems. In *ECRTS' 12.* IEEE Computer Society, Washington, DC, USA, 331–340.

[28] 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 (DAC '08).* ACM, New York, NY, USA, 300–303. https://doi.org/10.1145/1391469.1391545

[29] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. 2013. Making Shared Caches More Predictable on Multicore Platforms. In *ECRTS' 13.* 157–167.

[30] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages.

[31] J. Xiao, S. Altmeyer, and A. Pimentel. 2017. Schedulability Analysis of Non-preemptive Real-Time Scheduling for Multicore Processors with Shared Caches. In *2017 IEEE Real-Time Systems Symposium (RTSS).* 199–208. https://doi.org/10.1109/RTSS.2017.00026

[32] J. Xiao, S. Altmeyer, and A. D. Pimentel. 2020. Schedulability Analysis of Global Scheduling for Multicore Systems with Shared Caches. *IEEE Trans. Comput.* (2020), 1–1. https://doi.org/10.1109/TC.2020.2974224

[33] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. [n. d.]. Holistic Resource Allocation for Multicore Real-Time Systems. In *RTAS'19.* https://doi.org/10.1109/RTAS.2019.00036

[34] M. Xu, L. T. X. Phan, H. Y. Choi, and I. Lee. 2016. Analysis and Implementation of Global Preemptive Fixed-Priority Scheduling with Dynamic Cache Allocation. In *RTAS.* 1–12.

[35] Maolin Yang, Wen-Hung Huang, and Jian-Jia Chen. 2018. Resource-Oriented Partitioning for Multiprocessor Systems with Shared Resources. *IEEE Trans. Comput.* PP (12 2018), 1–1. https://doi.org/10.1109/TC.2018.2889985

[36] W. Zhang and J. Yan. 2009. Accurately Estimating Worst-Case Execution Time for Multi-core Processors with Shared Direct-Mapped Instruction Caches. In *RTCSA '09.* 455–463.