Cache Interference-aware Task Partitioning for Non-preemptive Real-time Multi-core Systems

JUN XIAO, YIXIAN SHEN, and ANDY D. PIMENTEL, University of Amsterdam, Netherlands

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 article 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. We first analyze the shared cache interference between two programs for set-associative instruction and data caches. Then, 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 and other greedy partition approaches such as First-fit and Worst-fit over randomly generated tasksets and realistic workloads in embedded systems. Our empirical evaluations show that CITTA outperforms global EDF scheduling and greedy partition approaches in terms of task sets deemed schedulable.

CCS Concepts: • **Computer systems organization → Embedded software;**

Additional Key Words and Phrases: Shared caches, partitioned scheduling, schedulability analysis, real-time systems

ACM Reference format:
Jun Xiao, Yixian Shen, and Andy D. Pimentel. 2022. Cache Interference-aware Task Partitioning for Non-preemptive Real-time Multi-core Systems. *ACM Trans. Embedd. Comput. Syst.* 21, 3, Article 28 (May 2022), 28 pages.
https://doi.org/10.1145/3487581

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 [29], 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

Authors' address: J. Xiao, Y. Shen, and A. D. Pimentel, University of Amsterdam, Amsterdam, Netherlands; emails: j.xiao@uva.nl, y.shen@uva.nl, a.d.pimentel@uva.nl.

1539-9087/2022/05-ART28 $15.00
https://doi.org/10.1145/3487581

is preempted and its data are 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 Reference [39], 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 article, 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). Hardy and Puaut [24] 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, however, 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 [7].

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, then 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 [43], 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).

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.

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

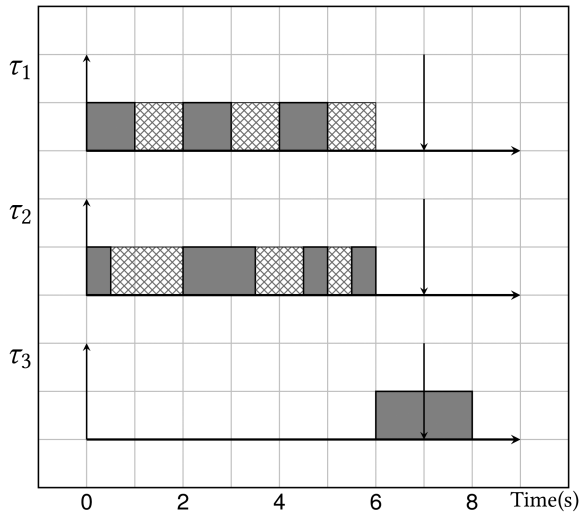


Fig. 1. Case where τ_3 misses its deadline if τ_1, τ_2, and τ_3 are scheduled globally.

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 original version of our cache interference-aware task partitioning algorithm for real-time multicore systems with shared caches was presented in Reference [45] for direct-mapped caches. Significant extensions are made in this article, including the following:

- We have extended the analysis on shared cache interference for set-associative instruction and data caches.
- We have implemented our analysis on shared cache interference between two programs and integrated it into the Heptane [25] WCET estimation tool.
- We have conducted experiments to obtain the worst-case cache interference among multiple applications using the Mälardalen WCET benchmark suite [22] and TACLeBench [16]. With these realistic workloads in embedded systems, we illustrate the advantage of CITTA over global scheduling.

The rest of the article is organized as follows. Section 2 gives an overview of related work. The system model and some other prerequisites for this article are described in Section 3. Section 4 is the extended analysis on shared cache interference between two programs for the set-associative instruction and data caches. Section 5 describes the proposed CITTA, where we also detail the computation of the inter-core cache interference and schedulability analysis of CITTA. Section 6 presents the experimental results, after which Section 7 concludes the article.

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 Reference [42]. 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 [23, 32, 49]. In almost all those works, due to the assumption that cache interference 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 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 [19]. The most common software-based cache partitioning technique is page coloring [33, 41]. By exploiting the virtual-to-physical page address translations present in virtual memory systems at the 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 [10, 37, 39], 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 [20, 47], 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.

Shared cache interference is analyzed in Reference [43], but that holds only for direct-mapped instruction caches. In this work, we extend the analysis to set-associative caches as well as to data caches.

Real-time Scheduling. To schedule real-time tasks on multi-core platforms, different paradigms have been widely studied: partitioned [5, 17, 48], global [4, 8, 31], and semi-partitioned scheduling [9, 12, 28]. A comprehensive survey of real-time scheduling for multiprocessor systems can be found in Reference [15]. 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 [46]. Guo et al. [21] address the problem of intra-core cache interference, which occurs within one core, when a task is preempted. They leverage the way-allocation technique to partition the last-level cache for individual cores to eliminate inter-core cache interference. However, our work addresses inter-core cache interference and uses non-preemptive scheduling to avoid the inter-task interference.

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 Reference [44], which also addresses schedulability analysis for multi-core systems with shared caches. However, the work of Reference [44] only considers

global scheduling. In this article, 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

3.1.1 Task Model. A taskset τ comprises n periodic or sporadic real-time tasks $\tau_1, \tau_2, \ldots, \tau_n$. 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 article 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 article are assumed to be non-negative integer values.

3.1.2 Architecture Model. Our system architecture consists of a multi-core processor with m identical cores onto which the individual tasks are scheduled. We assume a fully timing compositional architecture without timing anomalies [35].

In multi-core processors, caches are organized as a hierarchy of multiple cache levels to address the tradeoff 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. Each cache implements the LRU replacement policy. We consider both set-associative instruction and data caches. Furthermore, caches are assumed to be non-inclusive non-exclusive, which means that (i) a memory block is searched for in cache level L (i.e., LLC) if and only if a cache miss occurred when searching it in cache level $L1$. (ii) When a cache miss occurs at cache level L, the entire cache line containing the missed information is loaded into cache level L. (iii) The modification issued by a store instruction goes through the memory hierarchy. If the written memory block is already present at cache level L, then a write action is performed, along with the update of the main memory. Otherwise, if the information is absent at cache level L, then this cache is left unchanged.

In hard real-time systems, it is common to avoid the usage of virtual memory to improve timing predictability. In this work, we assume that a real-time task is compiled as a single binary and its physical memory address space is determined offline, before its execution. All real-time tasks directly use physical addresses. As the LLC in modern processors are typically physically indexed and physically tagged, the mapping between a memory block and the cache set where the block is stored can be derived.

The problem we are addressing in this article originates from the cache sharing present in the multi-core architectures. We illustrate the problem of cache interference by an example shown in Figure 2. Task A and Task B are scheduled on two different cores. During the execution, Task

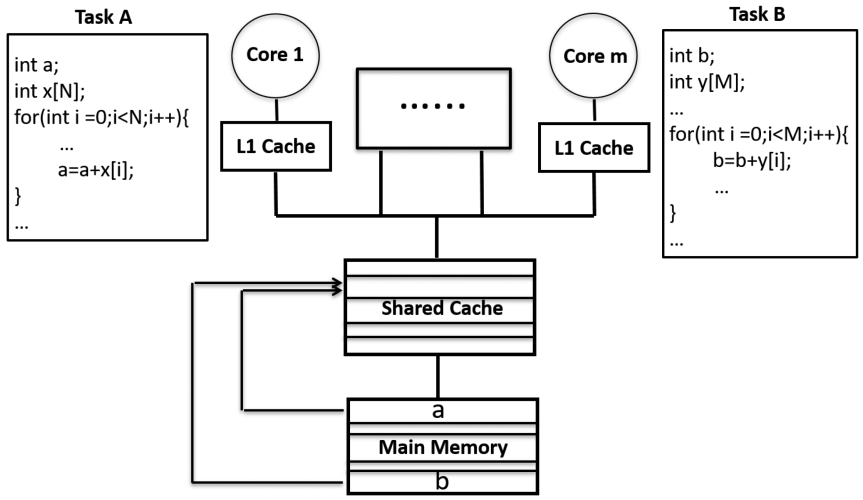


Fig. 2. The architecture model and the problem of cache interference.

A and B access their own variable a and b, which are stored in different locations in the main memory. When deriving the WCET of task A (or B), we assume there are no other tasks running simultaneously. The first access to a (or b) is considered to be a cache miss, while later accesses to a (or b) could be hits in the shared cache if the data are evicted in the lower-level caches by the execution of task A (or B) but not evicted in the shared cache. However, cache interference could occur if task A and B execute concurrently and the two memory allocations storing a and b map to the same cache set in the shared cache. In this case, the data a (or b) cached by task A (or B) previously gets evicted due to the assess to b (or a) during the execution of task B (or A). Consequently, a (or b) is loaded from main memory instead of from the shared cache, causing an extra delay for task A's (or B's) actual execution.

3.1.3 Partitioned Non-preemptive Schedulers. In this article, we study non-preemptive partitioned scheduling to avoid the analysis of cache related preemption delays, which has been intensively studied in References [2, 30, 36]. Once a task instance starts execution, any preemption during the execution is not allowed, it must run to completion. If not explicitly stated, cache interference will therefore solely refer to inter-core cache interference in the following discussion. However, the approach discussed in this work is not limited to non-preemptive real-time systems and can also be applied to preemptive systems with shared caches by taking the cache related preemption delays into account. We plan to extend our approach to preemptive scheduling as future work.

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 [18, 26] of non-preemptive EDF (EDF_{np}) makes it a reasonable algorithm to use as the runtime 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 runtime 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 [6]. 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 $t_0 + T_i, t_0 + 2T_i, t_0 + 3T_i, \ldots$ Therefore, $DBF(\tau_i, t)$ can be computed by Equation (0.1),

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

Reference [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:

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

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

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

$$DBF^*(\tau_i, t) \geq DBF(\tau_i, t). \tag{0.3}$$

3.3 Uniprocessor Schedulability

The schedulabity analysis of uniprocessor scheduling is well studied. References [3, 27] 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*

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

and for all $\tau_j \in \tau, \forall i \leq j, T_i \leq T_j$:

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

Note that the computation of $DBF(\tau_i, t)$ and $DBF^*(\tau_i, t)$ by Equations (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.

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 [42]. The CAC categorizes the accesses to a certain cache level as **Always (A)**, **Uncertain (U)**, **Never (N)**, or **Uncertain Never (UN)**. CHMC classifies the reference to a memory block as **Always Hit (AH)**, **Always Miss (AM)**, **First Miss (FM)**, or **Not-classified (NC)**. Table 1 describes CAC and CHMC classification terms.

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,

Table 1. Description of CAC and CHMC Analysis

Notation	Description
CAC	Classifies the references (abbreviated as r) used for the analysis at every cache level:
A	the access to r is always performed at cache level L
N	the access to r is never performed at cache level L
UN	the first access to r is unsure but next accesses are never performed at cache level L
U	the access to r is uncertain at cache level L
CHMC	Classifies the access state of access r to a memory block:
AH	the access r is guaranteed to be in cache level L
AM	the access r is guaranteed not to be in cache level L
FM	the access r is not guaranteed to be in cache the first time it is accessed, but is guaranteed afterwards
NC	the access r is not guaranteed to be in cache and is not guaranteed not to be in cache
HB	a memory block whose access is classified as AH at the shared LLC cache
CB	a memory block whose access is classified as A, U or UN at the shared LLC cache

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.

$I^c_{i,k}$ is bounded for direct-mapped instruction caches, as indicated by Lemma in Reference [43]. In this work, we extend the analysis of the cache interference for set-associative instruction and data caches.

4.1 Cache Interference Analysis for Set-associative Caches

Xiao et al. [43] introduced the concept of **Hit Block (HB)**, i.e., a memory block whose access is classified as *AH* or *FM* at the shared cache and the concept of **Conflicting Block (CB)**, i.e., a 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 formally describe how cache interference is calculated.

Given the source code of a program, we first generate its **control flow graph (CFG)**. For each basic block in the CFG, CHMC, and CAC is applied to the low-level analysis of instruction and data addresses. We use $HB^k = \{m^k_1, m^k_2, \ldots, m^k_p\}$ to represent the set of *HB* for task τ_k. Furthermore, we denote $age(m_x)$ as the age of a memory block m_x in the LRU stack, which is also one of the outcomes of CHMC analysis. Similarly, we define $CB^i = \{m^i_1, m^i_2, \ldots, m^i_q\}$ as the set of *CB* for task τ_i that are classified as an A, U or UN at the *LLC* cache. Note that *HB* and *CB* include all the basic blocks in every program path that may be considered by the tasks.

In our system architecture, cache interference occurs only at the shared *LLC* cache when a cache line used by τ_k is evicted by τ_i and consequently causing reload overhead for τ_k. A cache line that may cause cache interference for τ_k needs to satisfy at least three conditions:

(1) access to that cache line will result in a cache hit at the *LLC* cache in WCET analysis of τ_k;
(2) the cache line may be used by τ_i;
(3) the sum of distinguished accesses from τ_k and τ_i is larger than the cache associativity.

The first condition implies that only accesses in HB^k may result in cache interference for τ_k, while the second condition indicates that accesses in CB^i by τ_i may interfere with τ_k. Furthermore, cache interference occurs only if τ_k accesses memory blocks in HB^k and τ_i access memory blocks in CB^i concurrently. The last condition for interference entails that the total number of distinguished memory accesses by τ_i and τ_k that maps to the same cache set, requires to be larger than the degree of associativity such that cache evictions actually could take place.

Assuming that the cache set index of the *LLC* ranges from 0 to $N-1$, we can divide HB^k into N subsets according to the mapping function idx that maps a memory address to the cache set index at the *LLC* as follows:

$$\hat{m}_u^k = \{m_x \in HB^k | idx(m_x) = u\}, (0 \leq u < N, u \in \mathbb{N}).$$

Similarly, we divide CB^i into N subsets by

$$\hat{n}_u^i = \{m_x \in CB^i | idx(m_x) = u\}, (0 \leq u < N, u \in \mathbb{N}).$$

We define the characteristic function of a set A that indicates membership of an element x in A as

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & otherwise \end{cases}.$$

Let N_u^i represent the number of accesses to the uth cache set by τ_i. It is bound by

$$N_u^i = \sum_{x=1}^{q} \chi_{\hat{n}_u^i}(m_x), m_x \in CB. \tag{1.3}$$

Cache interference can only happen among memory blocks that map to the same cache set. For the uth cache set, τ_k can be interfered if the sum of the age of a memory block in HB^k and the total accesses from τ_i is larger than the cache associativity. We use MI_u^k to represent the set of memory blocks, by whose accesses that might be interfered at the uth cache set by task τ_i. It is calculated by

$$MI_u^k = \{m_x | m_x \in \hat{m}_u^k, age(m_x) + N_u^i > N_{aso}\}.$$

The total number of accesses to each memory block in MI_u^k is given by the number of iterations performed at the basic block. One can obtain the maximum number of iterations from the source code annotations provided by the static analysis with a WCET analysis tool, as will be explained in the next section. We use $A_{i,u}$ as the bound on the accesses to $m_i \in MI_u^k$.

The following formula gives an upper bound on the number of cache misses of accesses in HB^k for task τ_k:

$$S = \sum_{u=0}^{N-1} \sum_{m_i \in MI_u^k} A_{i,u}. \tag{1.4}$$

Suppose the penalty for an *LLC* cache miss is a constant, C_{miss}, then $I_{i,k}^c$ satisfies

$$I_{i,k}^c = S \times C_{miss}. \tag{1.5}$$

The computation only takes the memory accesses of τ_k and τ_i as input, so $I_{i,k}^c$ only depends on memory access of τ_k and τ_i. Therefore, the following Lemma holds:

Lemma 1. *$I_{i,k}^c$ can be bounded.*

4.2 Implementation of Cache Interference Analysis with Heptane

Heptane [25] is an open-source static WCET analysis tool. It has a special focus on analysis of cache hierarchies with multiple replacement policies, and it currently supports both MIPS and ARM v7 instruction sets.

As illustrated in Figure 3, given the source code of a pair of interfered program (P_{ed}) and interfering program (P_{ing}), written in C or assembly language, Heptane first calls the compiler and linker to generate a binary file and then constructs the CFG. It also identifies the different loops and attaches the loop bounds information provided in the source files of P_{ed} and P_{ing} programs.

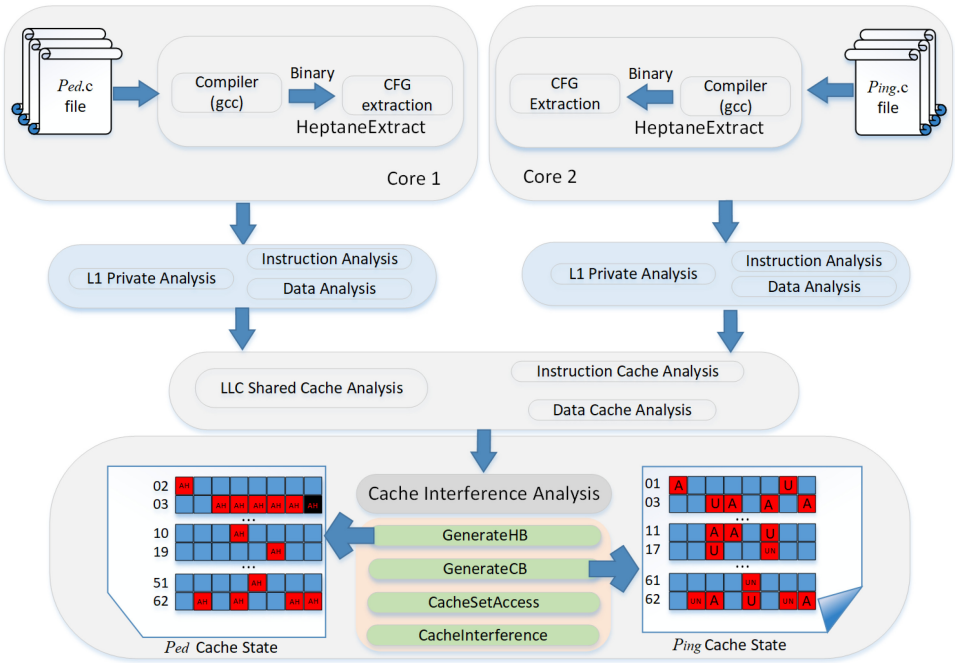


Fig. 3. The architecture of the Cache Interference Framework.

The Must, May, and Persistence analyses [40] based on abstract interpretation [14] is performed at each cache level for both instruction and data cache. The purpose of the analyses is to determine the CHMC classification for every memory reference.

We have extended Heptane with our cache interference computation as presented in the previous section. Our implementation performs a separate and sequential analysis for each level of caches in the memory hierarchy. As mentioned before, since the L1 cache is private to the cores and we consider non-inclusive caches, there is no cache interference at L1 caches. We only need to calculate cache interference at the *LLC*.

The core of the cache interference calculation consist of four functions: *GenerateHB*, *GenerateCB*, *CacheSetAccess* and *CacheInterference*. *GenerateHB* computes the *HB* for the interfered task τ_k. *GenerateCB* generates the *CB* for the interfering task τ_i. Given the memory blocks in *HB* and *CB*, *CacheSetAccess* calculates the distinguished memory blocks in *HB* for each cache set and the total number of accesses to each cache set from τ_i. Finally, *CacheInterference* computes the upper bound on cache interference exhibited by τ_k.

To validate our implementation, we have performed a comprehensive analysis of our extended version of Heptane using small benchmarks with predicable sequences of instruction and data cache accesses.

5 CACHE INTERFERENCE AWARE TASK PARTITIONING: CITTA

Given a taskset τ composed of n periodic or sporadic tasks and a processing platform π with m identical cores $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$, 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.

References [34] and [17] 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 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, then 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 5.1, we describe our cache interference-aware task partitioning: CITTA. Section 5.2 derives the calculation of the upper bound on the shared cache interference. Section 5.3 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 that accounts for cache interference. $DBF^c(\tau_i, t)$ can be computed by extending Equation (0.1):

$$DBF^c(\tau_i, t) = max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \times \left(C_i + \bar{I}_i^c\right)\right). \tag{1.6}$$

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

$$DBF^{c*}(\tau_i, t) = \begin{cases} 0 & t < D_i \\ C_i + \bar{I}_i^c + U_i^c \times (t - D_i) & otherwise \end{cases}, \tag{1.7}$$

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

It can also be observed that

$$DBF^{c*}(\tau_i, t) \geq DBF^c(\tau_i, t). \tag{1.8}$$

5.1 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 6.

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 $\tau_k \in \tau(\pi_x) \cup \{\tau_i\}$

$$D_k \geq \sum_{\substack{\tau_j \in \tau(\pi_x) \cup \{\tau_i\} \\ D_j \leq D_k}} DBF^{c*}(\tau_j, D_k) + \max_{\substack{\tau_j \in \tau(\pi_x) \cup \{\tau_i\} \\ D_j > D_k}} C_j + \bar{I}_j^c. \tag{1.9}$$

(a) If no τ_k violates condition (1.9), then the attempt is *admitted* and τ_i is added to $\tau(\pi_x)$.
(b) If condition (1.9) is violated by at least one τ_k, then 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, then 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, then 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 (1.9). 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 (1.9). 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, then the task τ_k will not miss its deadline. We will prove this in Section 5.3.

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 $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), \ldots, \tau(\pi_m)$. 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=0
```

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 $\tau_k \in \tau(\pi_x)$, i.e., tasks that are already assigned to π_x, is recomputed.

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

```
1: taskAssigned ← false, τ^tna ← ∅
2: for all τ_i ∈ τ do
3:    assignTo ← NULL, coreSuccess ← true
4:    for all π_x ∈ π do
5:       for all τ_k ∈ τ(π_x) ∪ {τ_i} do
6:          calculate Ī_k^c
7:       end for
8:       for all τ_k ∈ τ(π_x) ∪ {τ_i} do
9:          if condition (1.9) violates for τ_k then
10:            coreSuccess ← false
11:            break;
12:         end if
13:      end for
14:      if coreSuccess then
15:         τ(π_x) ← τ(π_x) ∪ {τ_i}
16:         assignTo ← π_x, taskAssigned ← true
17:         break;
18:      end if
19:   end for
20:   if assignTo == NULL then
21:      τ^tna ← τ^tna ∪ {τ_i}
22:   end if
23: end for
24: return τ^tna, taskAssigned, τ(π) = 0
```

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.

5.2 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 $\bar{I}_k^c$.

Reference [43] presented an approach to calculating the upper bound on cache interference for tasks that are globally scheduled. By extending the approach in Reference [43], 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 $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), \ldots, \tau(\pi_m)$ 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.

5.2.1 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 jth 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.

The objective function of the *IP* formulation is to maximize the total cache interference exhibited by task τ_k. If $N_{i,k}$ jobs of τ_i are executing concurrently with τ_k, then the cache interference that τ_i causes on τ_k is bounded by $N_{i,k} \cdot I^c_{i,k}$. 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 as follows:

$$\max \sum N_{i,k} \cdot I^c_{i,k}. \tag{1.10}$$

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, then, 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:

$$\forall \tau_i \in \tau(\pi_x), N_{i,k} = 0. \tag{1.11}$$

$N_{i,k}$ 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:

$$\forall \tau_i \notin \tau(\pi_x), \left\lfloor \frac{\max(0, C'_k - T_i)}{T_i} \right\rfloor + \xi_i \leq N_{i,k}, \tag{1.12}$$

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 $N_{i,k}$:

$$\forall \tau_i \notin \tau(\pi_x),\ N_{i,k} \leq 1 + \left\lceil \frac{\max(0, C'_k - T_i + D_i)}{T_i} \right\rceil. \tag{1.13}$$

If $N_{i,k} > 2$, then 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, then 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 π_y (with $\pi_y \neq \pi_x$) 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 (1.14):

$$\forall y \neq x, \sum_{\tau_i \in \tau(\pi_y) \cup \tau^{na}} \max(0, N_{i,k} - 2) C_i \leq C'_k. \tag{1.14}$$

The objective function (1.10) together with constraints on $N_{i,k}$, i.e., inequalities (1.11), (1.12), (1.13), and (1.14), 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: $\tau(\pi) = (\tau(\pi_1), \tau(\pi_2), \ldots, \tau(\pi_m)$, 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, then 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 2. *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: We show the proof sketch.

From condition 1.9, 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 2 is the reason CITTA forces the recalculation of upper bound on cache interference caused by tasks that are already assigned to cores by CITTA.

5.2.2 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 3. *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 (1.14) 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 $O((4n + m)64^n \ln 4n + m)$ [13].

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 $O(\gamma(4n^2 m + nm^2)64^n ln4n + m)$. Since the while loop in *CITTA* executes at most n times, the complexity of CITTA is $O(\gamma(4n^3 m + m^2 n^2)64^n ln4n + m)$.

5.3 Schedulability Analysis

5.3.1 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*

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

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

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

5.3.2 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 4. *If tasks are assigned to cores by CITTA, then*

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

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 (1.9), we have

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

From Equation (1.7), $DBF^{c*}(\tau_i, D_u)$ 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 5. *If the tasks are assigned to cores by CITTA, then $\forall \pi_x \in \pi$, $\tau(\pi_x)$ 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 (1.9), 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 $\exists t_f, C_p + \bar{I}_p^c \le t_f \le D_p$, 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 $D_i \le t_f$. Assuming $\forall \tau_i \in \tau(\pi_x)$, $D_i > t_f$, from Equation (1.7),

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

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 $D_i \le t_f$. Let τ_s be the task with the largest relative deadline, i.e., $D_s = max\{D_i | \tau_i \in \tau(\pi_x) \wedge D_i \le t_f\}$

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

From Inequalities (1.8) 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

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

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

τ_s satisfies condition (1.9):

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

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

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

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

This contradicts to Lemma 4.

(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 $D_s \le t_f \le D_p$. 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 $D_s < D_p$, then we have

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

and

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

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

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

Since τ_s satisfies condition (1.9),

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

From Inequalities (2.10) and (2.11),

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

This also contradicts to Lemma 4. □

The correctness of Algorithm CITTA follows, by application of Lemma 5:

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.*

6 EXPERIMENTS

We assess 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 non-preemptive EDF (GEDF)**, which is proposed in Reference [44], the only, at least to the best of our records, work on real-time multiprocessor scheduling taking the shared cache interference into account. Moreover, we also compare CITTA against other greedy partition algorithms (First-fit, Worse-fit) in the context of non-preemptive scheduling.

As mentioned in the beginning of Section 5.1, the CITTA algorithm first sorts tasks in non-decreasing order using some criterion and then assigns tasks to the processor cores according to Equations (1.9).

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.

We first conduct systematic evaluation to compare the performance of CITTA with global scheduling using randomly generated workloads, after which we illustrate the advantage of CITTA by taking an example of realistic workloads in embedded systems.

6.1 Systematic Evaluation

6.1.1 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 20,000 tasksets in each experiment.

As the task generation policies may significantly affect experimental results, we give the policies used in the experiments as follows.

Task utilization generation policy. We use Randfixedsum [38] to generate vectors that consist of N elements and whose components sum to the U_{tot}. Each element in the vector is assigned an individual task utilization U_k in the taskset.

Task period and WCET generation policy. For each task τ_k, T_k is uniformly distributed over the interval $[100, 200]$. The WCET of τ_k is derived by $C_k = T_k \times U_k$. We consider an implicit deadline task system, which implies that $D_k = T_k$.

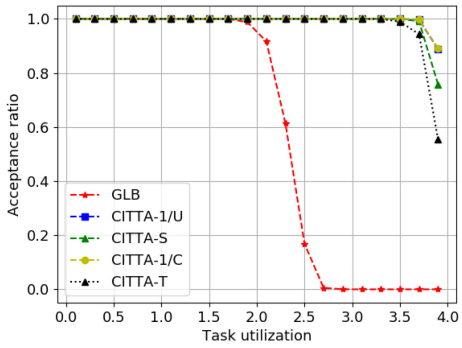
Cache interference generation policy. The probability of two tasks having cache interference is P. If two tasks τ_k and τ_i interfere with each other, then $I^c_{i,k}$ is generated as $I^c_{i,k} = IF \times min(0.5C_i, 0.5C_k)$.

To evaluate the schedulability performance of CITTA versus *GEDF*, 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*. Similarly, we select the same criteria to verify the performance of CITTA against the First-fit and Worst-fit partition algorithms. The acceptance ratio is the number of schedulable tasksets divided by the total number of tasksets.

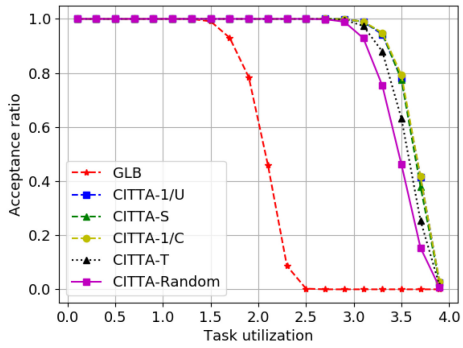
6.1.2 Results. We report the major trends characterizing the experimental results, illustrated in Figures 4, 5, and 6. In the figures, CITTA-<*criterion*> represents a variant of CITTA using <*criterion*> for sorting tasks, whereas **global scheduling (GLB)** stands for the GEDF scheduler.

CITTA outperforms global EDF. Our results clearly show that CITTA outperforms global EDF in all the test cases.

Figure 4 compares the acceptance ratio of CITTA and GLB when no cache interference exists in the system, i.e., $IF = 0$ and $P = 0$. It is clear that CITTA, as a partitioned scheduler, is more efficient than global scheduling.

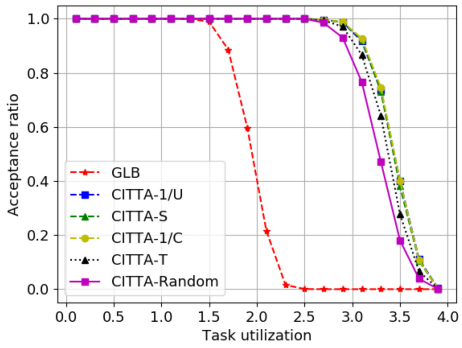


(a) m=4, n=10.

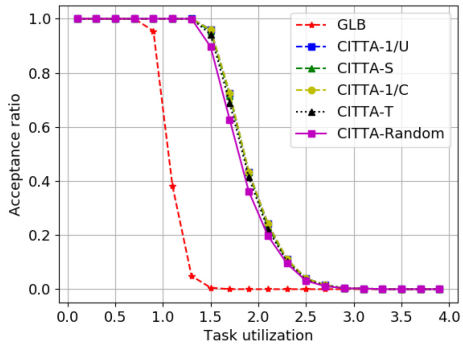


(b) m=4, n=20.

Fig. 4. Acceptance ratio without cache interference: $IF = 0, P = 0$.

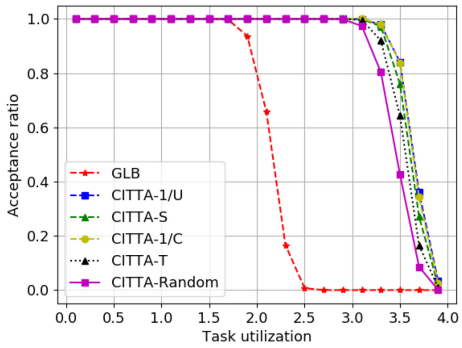


(a) IF=0.2, P=0.1.

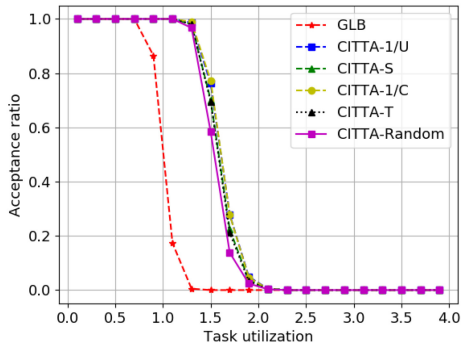


(b) IF=0.4, P=0.8.

Fig. 5. Acceptance ratio with different IF and P when $m = 4$, $n = 10$.



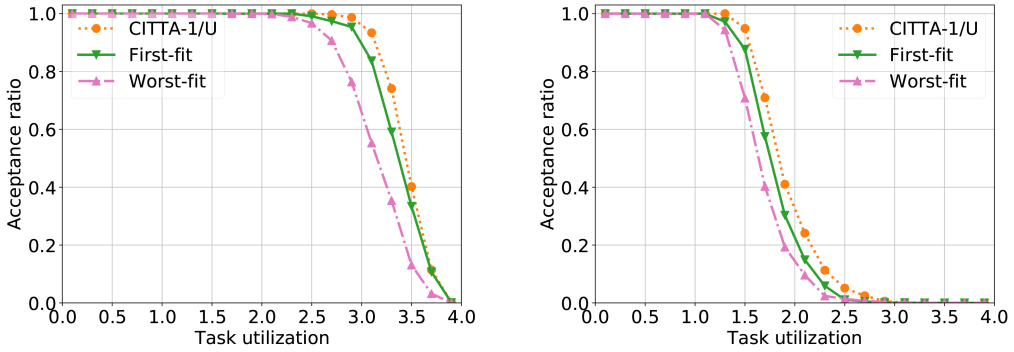
(a) IF=0.2, P=0.1.



(b) IF=0.4, P=0.8.

Fig. 6. Acceptance ratio with different IF and P when $m = 4$, $n = 20$.

When the degree of cache interference is generated by $IF = 0.2, P = 0.1$, as shown in Figure 5(a), 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 $U_{tot} = 2.5$.



(a) IF=0.2, P=0.1,m=4,n=10. (b) IF=0.4, P=0.8,m=4,n=10.

Fig. 7. Acceptance ratio with CITTA against First-fit and Worst-fit partition algorithm.

If cache interference is small, then the gap of acceptance ratio between all variants of CITTA and GLB is large for $U_{tot} \in [2.5, 3.5]$. From Figures 4(a), 5(a), and (b), when cache interference increases, the advantage of CITTA is less than GLB, which can be caused by the pessimism of the cache interference analysis. However, the schedulability performance gap still exists even when the cache interference is large, e.g., $IF = 0.4, P = 0.8$.

The comparison between Figures 4 and 5 clarifies that the schedulability benefit of CITTA does not come much from CITTA's cache analysis itself but mainly from the partitioned scheduling approach used by CITTA. However, this does not hurt the contribution of the proposed work, since, without CITTA, partitioned scheduling cannot be used safely and effectively in the presence of inter-core cache interference unless cache partitioning is used.

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 5(a) and 6(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 Figures 5(b) and 6(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.

Cache interference degrades schedulability performance. Figures 5(a) and 5(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. A similar observation can be made from Figures 6(a) and 6(b) for tasksets consisting of 20 tasks.

CITTA outperforms the greedy partitioned algorithms (First-fit, Worse-fit) as well. As illustrated in Figure 7, we observe that CITTA with the sorting criterion of the reciprocal of a task's utilization $1/U_i = T_i/C_i$ outperforms the first-fit and worse-fit partition algorithms with different degrees of cache interference. This is due to the fact that CITTA employs τ_{tna} to collect

Table 2. WCETs, Periods and Utilization of the 10 Selected Tasks in TacleBench

Name	Description	WCET (Cycles)	Period (Cycles)
expint	Series expansion for computing an exponential integral function	630,291	1,200,000
statemate	Automatically generated code	242,220	1,300,000
nsichneu	Simulate an extended Petri net	408,567	1,200,000
countnegative	Counts negative and non-negative numbers in a matrix	368,490	1,200,000
deg2rad	Conversion of degree to radiant	96,600	900,000
jfdctint	Discrete Cosine Transform on a 8×8 pixel block	116,291	800,000
minver	Matrix inversion for 3×3 floating point matrix	131,740	900,000
rad2deg	Conversion of radiant to degree	96,588	1,300,000

the unschedulable tasks and then forces an iterative recalculation of the upper bound cache interference of these unschedulable tasks to obtain an upper bound that is as small as possible, thereby achieving better schedulability performance.

6.1.3 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 2.4 GHz. On average, it takes 0.85 seconds to run CITTA for assignment of the taskset consisting of 10 tasks to a processor with four cores, while it takes 2.3 seconds for tasksets with 20 tasks.

6.2 A Closer Look at CITTA: A Case Study

We now take a closer look at selected, representative workloads from the embedded systems domain to better understand how CITTA performs better than global scheduling as well as the impact of cache interference on system schedulability.

We analyze the scenario where eight periodic tasks are to be scheduled on a multi-core processors with m cores.

Benchmarks. The workload is composed of three programs from Mälardalen WCET benchmarks [22], namely *expint*, *statemate*, and *nsichneu*, and five programs from the TACLeBench benchmark suite [16], e.g., *countnegative*, *deg2rad*, *ifdctint*, *minver*, and *rad2deg*. A brief description of the selected programs is provided in Table 2.

Architecture. We consider an embedded ARM processor with two cores. The cache hierarchy is composed of a 4-way L1 private cache with a cache line size of 32 B and an 8-way shared L2 cache with a cache line size of 64 B. The size of the L1 and L2 caches is 8 KB and 1 MB, respectively. The access latency of L1 cache, L2 cache and main memory is assumed to be 1, 10, and 100 cycles, respectively.

We first derive the WCET for the eight selected programs using the Heptane tool [25], targeting the ARM architecture. Since we consider periodic tasks, we determine the periods for the selected tasks in such a way that the task utilizations are in the range of $[5\%, 60\%]$. The derived WCETs, periods and task utilization without cache interference of the eight selected programs are listed in Table 2. Note that a task's WCET and period are measured in clock cycles.

As mentioned previously, Heptane is extended with the implementation of our analysis of cache interference between two programs. Table 3 lists the derived cache interference measured in clock cycles using our extended tool. Note that τ_k denotes the interfered task while τ_i is the interfering task.

Schedulability analysis. Given the task parameters and the underlying execution platform, we now perform the schedulability analysis to check whether the taskset is schedulable by global scheduling and CITTA.

Table 3. The Cache Interference between Two Programs, Measured by Cycles (TacleBench)

τ_k \ τ_i	*countnegative*	*deg2rad*	*expint*	*jfdctint*	*minver*	*nsichneu*	*rad2deg*	*statemate*
countnegative	—	13,300	3,100	10,900	3,500	11,000	13,300	11,200
deg2rad	5,800	—	900	2,600	2,000	5,400	73,300	6,800
expint	16,400	900	—	15,900	9,200	14,500	800	11,600
jfdctint	4,500	1,800	5,600	—	17,700	23,900	1,900	21,100
minver	6,700	2,200	7,900	23,300	—	28,900	2,300	27,700
nsichneu	24,600	12,000	31,000	123,600	95,300	—	11,900	182,200
rad2deg	5,800	73,400	1,000	2,600	2,000	5,400	—	6,800
statemate	11,100	6,800	11,000	60,400	56,900	86,400	6,700	—

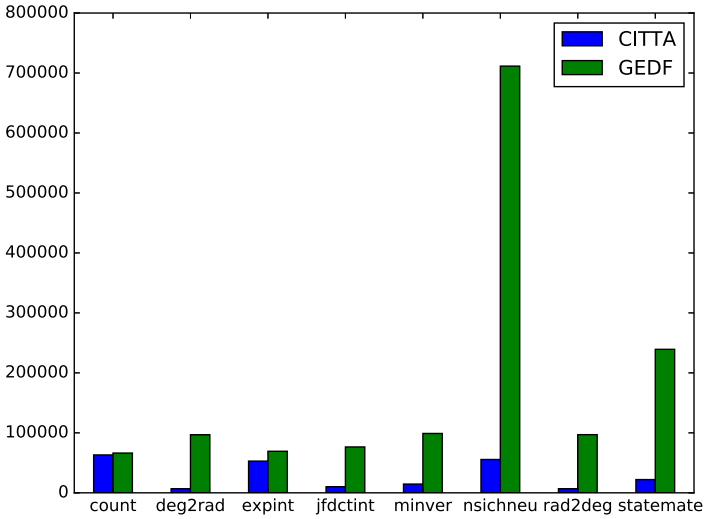


Fig. 8. Comparison of cache interference (measured in cycles) exhibited by each task under GEDF and CITTA.

Global scheduling. It can be verified that the taskset is not schedulable under GEDF by checking the schedulability condition of GEDF, proposed in Reference [44].

Partitioned scheduling. The taskset is scheduleable by $CITTA-1/U$. As the outcome of $CITTA-1/U$ partitioning algorithm, tasks *countnegative* and *expint* are assigned to core 0 and the remaining tasks are assigned to core 1.

Comparison of cache interference between GEDF and CITTA. The difference in schedulbility performance between GEDF and CITTA comes from the amount of cache interference among the tasks. Figure 8 compares the cache interference exhibited by each task in the taskset under GEDF and CITTA. As can be seen from Figure 8, *countnegative* exhibits the same interference under both GEDF and CITTA. When switching from GEDF to CITTA, the cache interference is reduced dramatically from 96,800 to 6,700, from 711,500 to 55,600, from 97,000 to 6,800, and from 239,300 to 22,100 for the tasks *deg2rad*, *nsichneu*, *rad2deg*, and *statemate*, respectively. This is due to the fact that the two task pairs, e.g., *deg2rad* and *rad2deg*, *statemate*, and *nsichneu* interfere heavily with each other under global scheduling. With CITTA, *deg2rad*, *nsichneu*, *rad2deg*, and *statemate* are executed on the same core, and hence it is not possible for them to interfere with each other.

Analysis time. We measure the analysis time taken by the Heptane tool for estimating tasks' WCET and our extended tool for calculating cache interference between two programs, as shown

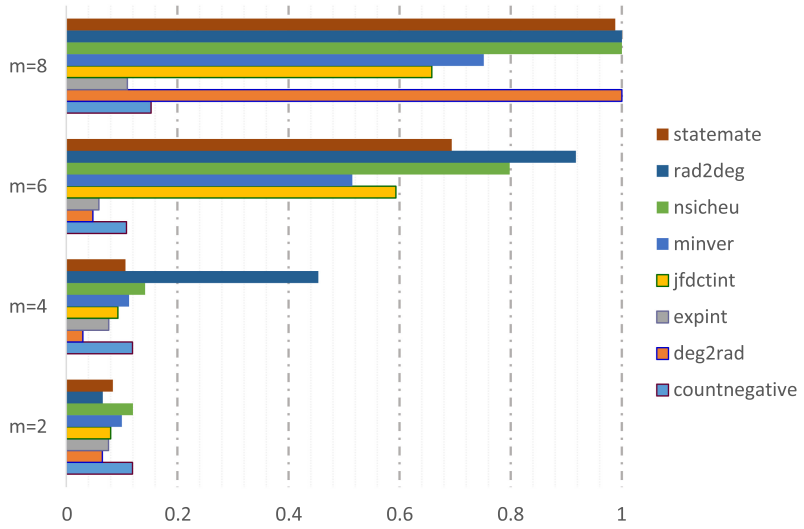


Fig. 9. Cache interference of eight periodic tasks with respect to the entire WCET by varying numbers of cores. The x-axis indicates the percentage of cache interference to tasks' WCET, and the y-axis shows four groups of experiments by varying the number of processing cores.

in Table 4. The analysis time is measured in seconds. The Heptane analysis time ranges from 0.05 seconds to 25 seconds and the extended analysis takes from 0.01 seconds to 1 seconds.

Pessimism of cache interference analysis. We evaluate the pessimism of the proposed approach to calculating cache interference by comparing the derived upper bound with tasks' WCET. When *expint* is interfered by *deg2rad*, the cache interference exhibited by *expint* is 0.014% of its WCET. The task *expint* exhibits the most interference when it is interfered by $ifdctint$, which accounts for 2.52% of its WCET. However, in some cases, the analysis is more pessimistic. For example, when *statemate* and *nsichneu* interfere each other, the upper bound on cache interference is 35.67% of the WCET for *statemate* and 44.59% for *nsichneu*.

Additionally, we evaluate the cache interference exhibited by each task when the workload consisting of eight periodic tasks execute on platforms with different number of processing cores. Figure 9 shows the percentage of cache interference to tasks' WCET. When the processing core is 2, the minimum, maximum, and average of cache interference concerning the entire execution time is 0.0649%, 0.12%, and 0.886% respectively. As the number of cores increases to 4, the cache interference of most tasks slightly increase. However, when the processing cores are more, i.e., $m = 8$, the system suffers from pessimistic cache interference: the minimum, maximum, and average cache interference is 10.99%, 100%, and 70.75%, respectively. When the cores are less, a task is interfered by less tasks. With the increasing number of cores, a task can be interfered by more tasks, which leads to accounting for more eviction from interfering tasks on the cache sets in *HB*.

Another factor that could influence the pessimism of the proposed schedulability analysis is the number of tasks in the task set. The proposed cache interference analysis calculates $I_{i,k}$ for each interfering task and simply aggregates $I_{i,k}$ from all interfering tasks without checking which cache blocks are evicted. This could lead to duplicate counting of the same evicted cache blocks and the upper bound on cache interference can be over-estimated. As for future work, we plan to develop methods to tighten the upper bound on cache interference.

Discussion on the integration of CITTA into a RTOS. The integration of CIITA into real-time operating systems such as $LITMUS^{RT}$ [11] involves two steps. The first step is to determine

Table 4. The Analysis Time of Heptane and Cache Interference(s)

τ_k \ τ_i	*countnegative*		*deg2rad*		*expint*		*jfdctint*	
	Heptane analysis	*Extended analysis*	*Heptane analysis*	*Extended analysis*	*Heptane analysis*	*Extended analysis*	*Heptane analysis*	*Extended analysis*
countnegative	0.121346	0.015748	0.121346	0.020486	0.121346	0.014998	0.121346	0.046885
deg2rad	0.076086	0.010398	0.076086	0.012172	0.076086	0.01188	0.076086	0.030758
expint	0.124165	0.017548	0.124165	0.01704	0.124165	0.015415	0.124165	0.038458
jfdctint	0.398735	0.044896	0.398735	0.043877	0.398735	0.046991	0.398735	0.065401
minver	0.823939	0.066835	0.823939	0.069955	0.823939	0.069974	0.823939	0.087381
nsichneu	24.163975	0.605423	24.163975	0.535292	24.163975	0.594377	24.163975	0.716172
rad2deg	0.075225	0.010308	0.075225	0.008084	0.075225	0.011574	0.075225	0.039327
statemate	4.175256	0.132362	4.175256	0.139073	4.175256	0.137318	4.175256	0.156777
τ_k \ τ_i	*minver*		*nsichneu*		*rad2deg*		*statemate*	
	Heptane analysis	*Extended analysis*	*Heptane analysis*	*Extended analysis*	*Heptane analysis*	*Extended analysis*	*Heptane analysis*	*Extended analysis*
countnegative	0.121346	0.060591	0.121346	0.466175	0.121346	0.01163	0.121346	0.09574
deg2rad	0.076086	0.050501	0.076086	0.470338	0.076086	0.010827	0.076086	0.099271
expint	0.124165	0.057069	0.124165	0.468208	0.124165	0.012944	0.124165	0.104446
jfdctint	0.398735	0.086509	0.398735	0.519875	0.398735	0.041931	0.398735	0.119838
minver	0.823939	0.107192	0.823939	0.529744	0.823939	0.067896	0.823939	0.149169
nsichneu	24.163975	0.558716	24.163975	0.919925	24.163975	0.533269	24.163975	0.670215
rad2deg	0.075225	0.054823	0.075225	0.499051	0.075225	0.011605	0.075225	0.098118
statemate	4.175256	0.170983	4.175256	0.607783	4.175256	0.15993	4.175256	0.180884

the task partitioning strategy. This is performed offline by CITTA, which is already exhibited and implemented in this work. The second step is to enforce the obtained partitioning scheme by setting the scheduling affinity of each task. Note that the schedulability analysis of CITTA assumes that tasks in each core are scheduled by non-preemptive EDF, this can be realized by setting the scheduler of each core to non-preemptive EDF. The second step is already supported by $LITMUS^{RT}$.

7 CONCLUSIONS

Shared caches in multi-core processors introduce serious difficulties in providing guarantees on the real-time properties of embedded software. In this article, 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. 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. We analyzed the shared cache interference between two programs for set-associative instruction and data caches. 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 and realistic workloads in embedded system. Our empirical evaluations shows that CITTA outperforms global EDF scheduling and other greedy partition approachs such as First-fit and Worst-fit 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 of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*. 187–195.

[2] Sebastian Altmeyer and Claire Maiza Burguière. 2011. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *J. Syst. Arch.* 57, 7 (2011), 707–719.

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

[4] Sanjoy Baruah. 2007. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'07)*. IEEE Computer Society, Washington, DC, 119–128.

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

[6] 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.

[7] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. 2010. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*. 14–24. https://doi.org/10.1109/RTSS.2010.23

[8] M. Bertogna, M. Cirinei, and G. Lipari. 2009. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Trans. Parallel Distrib. Syst.* 20, 4 (April 2009), 553–566. https://doi.org/10.1109/TPDS.2008.129

[9] B. B. Brandenburg and M. Gül. 2016. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'16)*. 99–110.

[10] Marco Caccamo, Marco Cesati, Rodolfo Pellizzoni, Emiliano Betti, Roman Dudko, and Renato Mancuso. 2013. Real-time cache management framework for multi-core architectures. In *Proceedings of the Real Time Technology and Applications Symposium (RTAS'13)*. IEEE Computer Society, Washington, DC, 45–54.

[11] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. 2006. Litmus^ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 111–126.

[12] 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 *Proceedings of the 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.

[13] Kenneth L. Clarkson. 1995. Las vegas algorithms for linear and integer programming when the dimension is small. *J. ACM* 42, 2 (1995), 488–499.

[14] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 238–252.

[15] 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

[16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis*.

[17] Nathan Fisher and Sanjoy Baruah. 2006. The partitioned multiprocessor scheduling of non-preemptive sporadic task systems. In *Proceedings of the 14th International Conference on Real-time and Network Systems*.

[18] Laurent George, Paul Muhlethaler, and Nicolas Rivierre. 1995. *Optimality and Non-preemptive Real-time Scheduling Revisited*. Research Report RR-2516. INRIA. Projet REFLECS.

[19] G. Gracioli and A. A. Fröhlich. 2013. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'03)*. 72–81.

[20] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *Proceedings of the 7th ACM International Conference on Embedded Software*. ACM, 245–254.

[21] Zhishan Guo, Kecheng Yang, Fan Yao, and Amro Awad. 2020. Inter-task cache interference aware partitioned real-time scheduling. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 218–226.

[22] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*, OpenAccess Series in Informatics (OASIcs), Björn Lisper (Ed.), Vol. 15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 136–146.

[23] D. Hardy, T. Piquet, and I. Puaut. 2009. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'09)*. 68–77.

[24] D. Hardy and I. Puaut. 2008. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'08)*. 456–466.

[25] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. 2017. The heptane static worst-case execution time estimation tool. In *Proceedings of the 17th International Workshop on Worst-Case Execution Time Analysis (WCET'17)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[26] K. Jeffay, D. F. Stanat, and C. U. Martel. 1991. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium*. 129–139.

[27] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. 1991. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the IEEE Real-time Systems Symposium*. IEEE, 129–139.

[28] S. Kato and N. Yamasaki. 2009. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. 23–32.

[29] H. Kim, A. Kandhalu, and R. Rajkumar. 2013. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'13)*. 80–89.

[30] Chang-Gun Lee, Hoosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.* 47, 6 (1998), 700–713.

[31] 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

[32] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. 2009. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*. 57–67. https://doi.org/10.1109/RTSS.2009.32

[33] J. Liedtke, H. Hartig, and M. Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *Proceedings of the Real Time Technology and Applications Symposium (RTAS'97)*. 213–224.

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

[35] Thomas Lundqvist and Per Stenstrom. 1999. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*. IEEE, 12–21.

[36] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. 2003. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 201–206.

[37] Mayank Shekhar, Abhik Sarkar, Harini Ramaprasad, and Frank Mueller. 2012. Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'12)*. IEEE Computer Society, Washington, DC, 331–340.

[38] Roger Stafford. 2006. Random vectors with fixed sum. (2006). http://www.mathworks.com/matlabcentral/fileexchange/9700.

[39] 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, 300–303. https://doi.org/10.1145/1391469.1391545

[40] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. 2000. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst* 18, 2–3 (2000), 157–179.

[41] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. 2013. Making shared caches more predictable on multicore platforms. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'13)*. 157–167.

[42] 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.

[43] J. Xiao, S. Altmeyer, and A. Pimentel. 2017. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'17)*. 199–208. https://doi.org/10.1109/RTSS.2017.00026

[44] 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

[45] 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*. 97–107.

[46] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *Proceedings of the Real Time Technology and Applications Symposium (RTAS'19)*. https://doi.org/10.1109/RTAS.2019.00036

[47] 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 *Proceedings of the Real Time Technology and Applications Symposium (RTAS'16)*. 1–12.

[48] 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

[49] W. Zhang and J. Yan. 2009. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*. 455–463.

Received December 2020; revised August 2021; accepted September 2021