TCPS: A Task and Cache-aware Partitioned Scheduler for Hard Real-time Multi-core Systems

Shen, Y.; Xiao, J.; Pimentel, A.D.

DOI
10.1145/3519941.3535067

Publication date
2022

Document Version
Final published version

Published in
LCTES '22

License
Article 25fa Dutch Copyright Act (https://www.openaccess.nl/en/in-the-netherlands/you-share-we-take-care)

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 426, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)
TCPS: A Task and Cache-Aware Partitioned Scheduler for Hard Real-Time Multi-core Systems

Yixian Shen
University of Amsterdam
Amsterdam, Netherlands
Y.Shen@uva.nl

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 seriously complicate the timing verification of real-time software tasks due to the task interference occurring in the shared caches. Explicitly calculating the amount of cache interference among tasks and cache partitioning are two major approaches to enhance the schedulability performance in the context of multi-core processors with shared caches. The former approach suffers from pessimistic cache interference estimations that subsequently result in suboptimal schedulability performance, whereas the latter approach may increase the execution time of tasks due to a lower cache usage, also degrading the schedulability performance.

In this paper, we propose a heuristic partitioned scheduler, called TCPS, for real-time non-preemptive multi-core systems with partitioned caches. To achieve a high degree of schedulability, TCPS combines the benefits of partitioned scheduling, relieving the computing resources from contention, and cache partitioning, mitigating cache interference, in conjunction with exploiting task characteristics. A series of comprehensive experiments were performed to evaluate the schedulability performance of TCPS and compare it against a variety of global and partitioned scheduling approaches. Our results show that TCPS outperforms all of these scheduling techniques in terms of schedulability, and yields a more effective cache usage and more stable load balancing.

CCS Concepts: • Computer systems organization → Embedded software.

Keywords: Cache partitioning, Partitioned scheduling, Schedulability analysis, Real-time systems

1 Introduction

Modern computer systems more and more rely on multi-core processors, in which an increasing number of cores are being integrated. These multi-core processors typically feature a Last Level Cache (LLC) that is shared among all cores. However, the shared LLC may easily hamper the real-time properties for safety-critical applications due to the resulting cache interference among concurrent application workloads. It is therefore crucial to provide timing predictable memory architectures for hard real-time systems where strict timing certification is mandated.

There are two main approaches for improving timing predictability of multi-core architectures, namely (1) explicitly calculating the possible cache interference such that it can be accounted for in the task scheduling and (2) cache partitioning to avoid the cache interference. In [38], the cache interference is explicitly modeled to obtain the worst-case execution time (WCET) and integrated into a global scheduler. This enhances real-time guarantees on multi-cores but suffers from pessimistic cache interference estimations that also result in suboptimal schedulability performance. Based on [38], [39] distributes tasks to individual cores and then employs a partitioned scheduler. However, this approach suffers from task allocation overheads and there still remains cache interference. To mitigate cache interference, IA3 [29] considers WCET-sensitivity to perform coarse-grained cache partitioning. However, it requires additional hardware support, does not exploit task characteristics for partitioning, and ignores the impact of monotonicity of WCET sensitivity (as will be discussed later on). [40] and [24] use cache partitioning and deploy a partitioned scheduler in the context of preemptive scheduling. However, in these works, the cache-related preemption and task reload overheads are assumed to be included in a task’s WCET by averaging its execution time over a number of runs rather than explicitly modeling and statically analyzing such overheads. Moreover, the technique...
in [40], which also applies memory bandwidth partitioning, was designed for soft real-time systems and cannot provide strong timing guarantees in hard real-time systems.

Despite the remarkable achievements on timing predictability in the presence of a shared LLC for real-time systems, only relatively little research attention has been paid to cache partitioning deployed in the context of non-preemptively scheduled multi-core platforms. Moreover, proper studies on the effectiveness of partitioned caches compared to non-partitioned caches in the context of global scheduling versus partitioned scheduling are missing.

In non-preemptive scheduling, cache-related preemption delays, which have been well studied [28], do not need to be considered. There are, however, three research challenges in realizing timing predictability on multi-cores through cache partitioning using non-preemptive scheduling: (1) how to ensure that real-time concurrent workloads safely obtain computing resources (e.g., considering the cumulative computing requirements for a specific taskset, taking into account so-called carry-in jobs [22], etc.), (2) how to effectively partition the shared cache space to the distinct cores, and (3) how to choose an appropriate real-time scheduler (i.e., partitioned versus global scheduler) to enhance the schedulability performance.

In this work, we propose a novel scheduler using a partitioned LLC with partitioned scheduling, which is holistically combined with a task-to-core mapping policy, to address the above challenges. We perform a partitioned Earliest Deadline First (EDF) strategy [5] and model the computing resource conflicts using the largest possible cumulative execution demand of all concurrent workloads (more details on this in Section 2). Subsequently, we experimentally investigate the relationship between the WCET of a task and the partitioned cache size using application code from real-time benchmarks. By exploiting this relationship, our scheduling policy can effectively allocate tasks with the same characteristics onto identical cores

\[ m \geq 2 \]

identical cores \( \pi = \{\pi_1, \pi_2, ..., \pi_m\} \). In this multi-core processor, caches are organized as a hierarchy of two cache levels where the lower level caches, i.e., L1 caches, are private while the last-level cache (LLC) is shared among all cores.

We consider both set-associative instruction and data caches and implement the LRU replacement policy for each cache level. Instruction caches and data caches are separate for L1 caches yet unified at the LLC level. Furthermore, caches are assumed to be non-inclusive non-exclusive, which means that: (i) A memory block is searched for in the 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, a write action is performed, along with the update of the main memory. Otherwise, if the information is absent at cache level L, this cache is left unchanged.

2 System Model and Prerequisites

2.1 System and Task Model

2.1.1 Architecture Model. The system architecture assuming a fully timing compositional architecture without timing anomalies consists of a multi-core processor with \( m \geq 2 \) identical cores \( \pi = \{\pi_1, \pi_2, ..., \pi_m\} \). In this multi-core processor, caches are organized as a hierarchy of two cache levels where the lower level caches, i.e., L1 caches, are private while the last-level cache (LLC) is shared among all cores.

We consider both set-associative instruction and data caches and implement the LRU replacement policy for each cache level. Instruction caches and data caches are separate for L1 caches yet unified at the LLC level. Furthermore, caches are assumed to be non-inclusive non-exclusive, which means that: (i) A memory block is searched for in the 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, a write action is performed, along with the update of the main memory. Otherwise, if the information is absent at cache level L, this cache is left unchanged.

2.1.2 Task Model. We consider a taskset \( \Gamma \) comprising \( n \) periodic or sporadic real-time tasks \( \Gamma = \{\tau_1, \tau_2, ..., \tau_n\} \). Each task \( \tau_i = (C_i, D_i, T_i) \in \Gamma \) is characterized by its bound worst-case execution time \( C_i \) obtained through the static analysis tool Heptane [21], a period or minimum inter-arrival time \( T_i \), and a relative deadline \( D_i \). In this paper, we assume constrained deadline tasksets which means that task relative deadlines are less or equal to the task period: \( D_i \leq T_i \). We further assume that all those tasks are independent, i.e., they have no shared variables, no precedence constraints, and so on.
A task \( \tau_j \) consists of a sequence of jobs \( f^j_i \), where \( j \) is the job index. We denote the arrival time, starting time, finishing time, and absolute deadline of a job \( j \) as \( r^j_i, s^j_i, f^j_i \) and \( d^j_i \), respectively. The primary goal of a real-time scheduling algorithm is to guarantee that each job will complete within its absolute deadline: \( f^j_i \leq d^j_i = r^j_i + D_i \).

### 2.2 Schedulability Verification

Cache-aware partitioned scheduling, in combination with cache partitioning, reduces multi-core scheduling into \( m \) groups of uniprocessor scheduling problems since the cache interferences among cores is completely avoided. Therefore, in this subsection, we recapitulate the exact schedulability verification for non-preemptive EDF scheduling of constrained-deadline tasks based on uniprocessor scheduling analysis [1]. We employ partitioned EDF for constrained deadline tasks onto distinct cores. Since there exists execution contention when multiple real-time tasks are ready to execute on a core, we model this contention through the Demand-Bound Function \( DBF(\tau_i, t) \) [8] to provide the guarantee to safely obtain the computing resources for each task. The \( DBF(\tau_i, t) \) function is often suggested to ensure a real-time task can be safely provisioned by the processor through accounting for the largest possible cumulative execution demand of all jobs generated by \( \tau_i \) equipped with both their arrival times and 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 jobs of \( \tau_i \) over \([t_0, t_0 + t] \) is maximized if one job arrives just at \( t_0 \) and subsequent jobs arrive as soon as permitted i.e., at instant \( t_0 + T_i, t_0 + 2T_i, t_0 + 3T_i, ... \). Therefore, \( DBF(\tau_i, t) \) can be computed by Equation (1).

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

Karsten et al. [1] used \( DBF(\tau_i, t) \) to model the processor contention process and exhibited a necessary and sufficient condition as shown in Theorem 1 for the feasibility test of a sporadic task system \( r \) scheduled by non-preemptive EDF scheduling, abbreviated as \( EDF_{np} \), on uniprocessor platforms.

**Theorem 1.** A taskset \( \Gamma \) is schedulable under \( EDF_{np} \) on a uniprocessor architecture if and only if:

\[
\forall t, \sum_{i=1}^{n} DBF(\tau_i, t) \leq t
\] (2)

and for all \( \tau_j \in \Gamma \):

\[
\forall t: C_j \leq t \leq D_j : C_j + \sum_{i=1 \& i \neq j}^{n} DBF(\tau_i, t) \leq t
\] (3)

Karsten et al. also [1] proposed an approach to approximate the \( DBF(\tau_i, t) \) using an approximated demand bound function \( DBF^*(\tau_i, t) \) :

\[
DBF^*(\tau_i, t) = \begin{cases} 0 & t < D_i \\ C_i + U_i \times (t - D_i) & \text{otherwise.} \end{cases}
\] (4)

where \( U_i = \frac{C_i}{T_i} \). We can observe that the following inequality holds for all \( \tau_i \) and all \( t > 0 \):

\[
DBF^*(\tau_i, t) \geq DBF(\tau_i, t)
\] (5)

In our work, we conduct schedulability verification of real-time tasks on each core by following Theorem 1. In order to simplify the calculation of the \( DBF^* \) function yet without overestimating, we replace \( DBF(\tau_i, t) \) with \( DBF^*(\tau_i, t) \). A schedulability test is sustainable [5] if a subset of tasks assigned to a dedicated core is deemed schedulable. If all tasks within the entire taskset pass the schedulability verification on \( m \) distinct cores, we derive that the taskset is schedulable on the multi-core architecture.

### 3 WCET Sensitivity to Cache Partitioning

Cache partitioning can avoid cache interference among cores through allocating cache space to individual cores while partitioned scheduling reduces task migration overheads and reduces \( DBF \) values for concurrent tasks due to fewer tasks executing on a single core. In this paper, we propose the TCPS scheduling approach that combines the benefits of cache partitioning and partitioned scheduling. Since it is not trivial to determine a good task-to-core allocation and an optimal per-core cache partitioning scheme, we first need to know the cache-space demand of tasks and then find the right trade-off between per-core cache space and task-to-core mappings. Hence, we begin by experimentally exploring the relationship between the WCET of an application and the varying size of its allocated cache partition (i.e., number of cache sets) in realistic benchmarks. By exploiting this relationship, we can determine which workloads feature similar characteristics in terms of cache-resource demands and, doing so, further guide the tasks allocation strategy and cache partitioning strategy.

As the experimental platform, we used ARMv7 processors with private 4-way set-associative L1 data and instruction caches of 1KB each. The LLC is 8-way set-associative and 64KB in size. With regard to the benchmarks, we used the Mälardalen [20] and TACLeBench [17] benchmark suites. We deployed the Heptane [21] framework to obtain the WCET values, which has a particular focus on cache analysis compared to other WCET analysis tools [36]. Both instruction and data caches were considered to evaluate the sensitivity of a task’s execution time to its allocated cache size. The results are shown in Figure 1.

In Figures 1a and 1b, each line denotes the normalized WCET for an individual benchmark with a varying number of allocated cache sets (i.e., partition sizes). We observe that the execution time of tasks decreases while increasing the
cache-partition size, up to a specific point which we refer to as the stable point. After this stable point, the execution time stabilizes and becomes insensitive to further increases of allocated cache-partition size. The benchmark tasks in Figure 1a consist of light-weight tasks. In most cases, the stable point is reached by only allocating 20% of cache space for these tasks. On the other hand, TACLeBench includes some more complex benchmarks, such as Lift (a lift controller program) that demands more cache sets to reach the stable point.

Most of the studied benchmarks follow the aforementioned stable point behavior, with a few exceptions. For instance, the benchmark nsichneu from the Mälardalen benchmark suite shows a higher execution time when it receives 40% of the cache sets compared to 20% of the cache sets. It is crucial that the WCET sensitivity to cache-partition size follows a strictly monotonic behavior to guide our cache and tasks allocation strategy. Otherwise, the allocation process may need to explore the full range of cache partitioning options, which is complicated and impractical. [2] proposed an approach to establish monotonicity for these counter-intuitive cases using the upper and lower bound of the sensitivity curve. We also perform this approximation approach to establish monotonicity for the counter-intuitive benchmarks without significant loss of precision. In Section 5, we evaluate the error introduced by this approximation by studying the schedulability performance for both the upper bound (i.e., conservative) and lower bound WCET values.

4 The TCPS Algorithm

In this section, we present our cache-aware partitioned scheduling algorithm, TCPS, for real-time multi-core systems. Driven by the monotonic execution-time property for realistic benchmarks, as discussed in the previous section, we distribute tasks to distinct cores based on the similarity of their cache space demands. Moreover, we perform the cache partitioning strategy, mapping the specific cache sets to individual cores, by finding the right trade-off among cores.

**Task allocation strategy.** Tasks allocated to the same core enjoy the same amount of partitioned cache memory. Hence, it is important to assign tasks that exhibit similar cache-demand characteristics to the same core to make full use of the assigned cache partition.

Each WCET sensitivity curve, as shown in Figure 1, can be modeled as a slowdown vector \( \vec{g}_i \). We assume set-associativity cache with \( S \) cache sets. The whole taskset \( \Gamma \) consisting of \( n \) tasks can be constructed as a vector \( \vec{G} \) comprised of \( n \) \( S \)-dimensional \( \vec{g}_i \), i.e., \( \vec{G} = (\vec{g}_1, \vec{g}_2, ..., \vec{g}_n) \). We exploit K-means clustering [26] to group the tasks with a similar slowdown vector \( \vec{g} \) for mapping to distinct cores. We employ the K-means clustering because it is suitable for our low dimensional data space, guarantees convergence, and has a relatively low overhead. We minimize the pairwise derivation in each cluster according to Equation 6 given a maximum number of iterations to converge, referred to as maxI.

\[
\arg\min_\theta \sum_{i=1}^{m} \frac{1}{2||\theta_i||} \sum_{r_s, r_j \in \theta_i} || \vec{g}_k - \vec{g}_j ||^2
\]  
(6)

Tasks are grouped into \( m \) clusters using K-means clustering. However, we cannot ensure that the tasks in each cluster are schedulable on a dedicated core since a task-to-CPU assignment purely based on the WCET sensitivity to partitioned cache size is sub-optimal. This may still lead to a situation in which the tasks mapped to a particular core (having similar WCET sensitivity to partitioned cache size) cause a core utilization of above 100%. Therefore, we adopt so-called taskbuckets \( \theta \) as temporal space and replicas for cores and fetch the tasks from taskbucket \( \theta_i \) to corresponding core \( \pi_i \) to conduct schedulability verification to enforce that
the utilization of each core will not exceed 100%. In addition to utilization-aware task partitioning, we also employ task migration in a later stage of our algorithm when the load across cores is not balanced.

**Cache partitioning strategy.** It is difficult to determine the best cache partitioning schema since allocating cache space to individual cores is similar to the bin-packing problem. Hence, we perform a heuristic cache partitioning strategy for cores in three stages by fully exploiting the slowdown vector $\vec{\tau}$ and the DBF value of each task on a dedicated core. Figure 2 illustrates the high-level idea of our proposed algorithm. The initial cache partitions, as will be detailed below, are based on the original taskbuckets that group tasks having similar WCET behavior in terms of allocated cache size. In the start-up cache partitioning phase, tasks are assigned based on their original taskbucket. If there exist any unschedulable tasks after this initial phase, the algorithm features two additional rounds of partitioning that include incrementally enlarging the cache partitioning of cores as well as task migrations between taskbuckets to reach schedulability. Algorithms 1 and 2 provide a detailed description of the TCPS algorithm.

1. **Start-up cache partitioning stage**, lines 1-3 in Algorithm 1. We implement the initial cache partitioning by first satisfying the least cache-demanding task of each core while maximizing the utilization of these tasks. More specifically, we select the least cache-demanding task from each $\theta_i$ (associated with core $\pi_i$), with $0 \leq i < m$. Based on the sensitivity curves for the $m$ selected tasks and the cache-space constraints of $m$ partitions, the objective function of a Mixed Integer Linear Programming (MILP) formulation $F$ as shown in equation 7 is constructed to find the cache partition size $p_i$ for each core $\pi_i$ that maximizes the total utilization of these $m$ tasks.

$$ F = \max \sum_{i=1}^{m} U_i = \max \sum_{i=1}^{m} \frac{C_i}{\tau_i} $$

With $\tau_i$ referring to the least cache-demanding task in the taskbucket $\theta_i$, we denote the utilization of $\tau_i$ as $U_i$, calculated by $\frac{C_i}{\tau_i}$. Inequality (8) specifies the constraint that the total of $m$ cache partitions $p_i$ must be less or equal to the total number of cache sets $S$.

$$ \sum_{i=1}^{m} p_i \leq S \quad (8) $$

Condition (9) consists of regression models for the $m$ tasks, capturing their WCET sensitivity curves, where $A_i$, $B_i$, and $k_i$ represent the regression parameters for fitting the WCET with varying $p_i$ of a task $\tau_i$.

$$ \sum_{i=1}^{m} C_i \times A_i + p_i \times B_i + k_i = 0 \quad (9) $$

The slope of each sensitivity curve is, however, not a constant. The challenge in implementing the whole partitioning problem as a MILP is to explore the sensitivity curve of all applications and generate a piece-wise linear fitting for tasks. Exploiting the continuous property of each sensitivity curve, we can linearize the piecewise function through Gurobi modeling [3]. Assuming there exist $q$ piece-wise functions on a sensitivity curve $f(C_i)$, with $C_i \leq C_{i+1} \leq \cdots \leq C_{iq}$, we use the continuous variable $w_i$ and 0-1 variable $z_i$ to fit each piece-wise function:

$$ C_i = \sum_{l=1}^{q+1} w_i \cdot C_{il} \quad (10) $$

$$ f(C_i) = \sum_{l=1}^{q+1} w_i \cdot f(C_{il}) \quad (11) $$

Variables $w_i$ and $z_i$ must satisfy the following constraints:

$$ \begin{align*}
 w_1 &\leq z_0, \quad w_2 \leq z_0 + z_1, \quad \cdots \quad w_q \leq z_{q-1} + z_q, \quad w_{q+1} \leq z_q \\
 w_1 + w_2 + \cdots + w_q + w_{q+1} &\leq 1 \\
 z_1 + z_2 + \cdots + z_q &\leq 1 
\end{align*} \quad (12) $$

After the start-up cache partitioning, each core has been allocated an initial cache partition. Subsequently, tasks are ordered by decreasing utilization in taskbuckets $\theta$ since it is typically harder for a task with high utilization to find an allocable core, whereafter we check whether or not all tasks are schedulable with this provided cache partition (lines 4-6 in Algorithm 1). To this end, we iteratively fetch all tasks from taskbucket $\theta_i$ to its corresponding core $\pi_i$, while verifying the schedulability using Algorithm 2. More specifically, we check if the condition (13) holds for each $\tau_k \in \tau_i$ on all $m$ cores.
We briefly explain the rationale behind the condition (13). Given task \( \tau_k \), \( \tau_j \) refer to the tasks that have been previously allocated to core \( \pi_i \). The leading term in condition (13) accounts for the cumulative execution demand of tasks (including \( \tau_k \)) with a relative deadline falling before \( D_k \). Since we consider a non-preemptive task system, the second term represents the maximum blocking time due to a task with a longer relative deadline than \( D_k \) at the time a job of \( \tau_k \) arrives. If the sum of these two terms is shorter than \( D_k \), the task \( \tau_k \) can safely finish within its deadline. If all tasks in all taskbuckets \( \theta \) fulfill condition (13), the taskset \( \Gamma \) is deemed schedulable, implying that the cache partitioning algorithm can be stopped. Otherwise, if there are still unused cache sets not allocated by the MILP, we turn to the next stage.

**ii) First-round cache partitioning stage**, lines 7-10 in Algorithm 1. The start-up cache partitioning stage only satisfies the least cache-demanding task on a core. Hence, in most cases, there remains sufficient unused cache space not allocated by the MILP. Therefore, we iteratively allocate a portion of cache space in units of \( \delta = 2 \) in our case) cache sets to distinct cores. After each iteration, we again take each task from taskbucket \( \theta_i \) and place it on corresponding core \( \pi_i \) while conducting the schedulability verification. Please note that if a new task is placed successfully, the tasks that have been previously allocated to core \( \pi_i \) are still schedulable since the increasing cache partition size of a core will result in a monotonic decrease of its total utilization. When taskbucket \( \theta_i \) is empty or all of its tasks reach the stable point, its associated core will stop receiving more cache space. The first-round cache partitioning stage ends when all taskbuckets are empty or have reached the stable point.

**iii) Second-round cache partitioning stage**, lines 11-23 in Algorithm 1. When the cache space is not used up after the first two stages, while there still exist unschedulable tasks in taskbuckets \( \theta \), we need to migrate tasks to other cores. However, the tasks in a more cache-demanding taskbucket can move to a less cache-demanding taskbucket, resulting in a longer execution time of these tasks. In this situation, we attempt to allocate more cache space to the target core if the tasks can be scheduled on that core after (re-)allocation.

We sort the cores \( \pi \) based on the utilization in increasing order since a core with lower utilization may accept more unschedulable tasks. Moreover, we define a set \( \Gamma^{ina} \) that contains the unschedulable tasks from taskbuckets \( \theta_i \), where \( k \neq i \) and \( \pi_i \) is the target core for migration. We iteratively allocate the cache sets to the target core and fetch the tasks from \( \Gamma^{ina} \) to the target core while conducting the scheduling verification. If all the unschedulable tasks can be allocated to the cores, the taskset is deemed schedulable.

\[
D_k \geq \sum_{\tau_j \in \{\tau_k\} \cup \{\pi_i\} \setminus \{\tau_k\}} DBF^s(\tau_j, D_k) + \max_{\tau_j \in \{\pi_i\} \setminus \{\tau_k\}} C_j \tag{13}
\]

We observe that a taskset can be unschedulable onto \( m \) cores but may become schedulable on fewer cores (as there is more cache space for each core). However, more tasks allocated to a dedicated core increase the DBF function as well as the utilization of that core. It is highly challenging to find an optimal number of cores. To achieve better schedulability with a minimum number of cores, we employ our TCPs algorithm to explore the feasible allocation on every valid number of cores in the range of \( 1 \leq i \leq m \). Hereafter, we adopt the smallest value of \( i \) to schedule that taskset.

**Algorithm 1** partitioned LLC for partitioned scheduling

**Input**: Task parameters, number of cores: \( m \), number of tasks: \( n \), sensitivity vector: \( \tilde{G} \), cache size: \( S \), maximum iterations: \( \text{maxI} \), the number of cache sets distributed at a time: \( \delta \)

1: TaskBuckets \( \theta \leftarrow \text{groupbySensitivity}(\tilde{G}, n, m, \text{maxI}) \\
2: Select the minimum vector \( G_{tb} = \{\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_m\} \) in \( \theta \) \\
3: \{\pi_1, \pi_2, \ldots, \pi_m\} \leftarrow \text{StartUpCachePartition}(G_{tb}, m, S) \\
4: Sort \( \tau \) in each \( \theta \) in decreasing order by utilization \\
5: if true \( == \) checkSchedulability(\( \theta, \pi \)) then
6: \( \text{return schedulable} \)
7: while \( \sum_{i=1}^m \{\pi_i + \delta\} < \delta \) do
8: \( \text{FirstRoundCachePartition}(\pi_1 + \delta, \pi_2 + \delta, \ldots, \pi_m + \delta) \)
9: if true \( == \) checkSchedulability(\( \theta, \pi \)) then
10: \( \text{return schedulable} \)
11: Sort the utilization of cores \( \pi \) in increasing order \\
12: for all \( \pi_i \in \pi \) do \\
13: Collect the unschedulable \( \tau \) in \( \theta \) to \( \Gamma^{ina}(\tau \neq \tau_0) \) \\
14: for all \( \tau_j \in \Gamma^{ina} \) do \\
15: if \( \sum_{i=1}^m \{\pi_i + \delta\} < \delta \) then \\
16: while \( \sum_{i=1}^m \{\pi_i + \delta\} < \delta \) do
17: \( \text{SecondRoundCachePartition}(\pi_1 + \delta) \)
18: if checkSchedulability(\( \Gamma^{ina}, \pi_i \)) \& (\( \tau_0 = 0 \)) then
19: \( \text{return schedulable} \)
20: else if checkSchedulability(\( \Gamma^{ina}, \pi_i \)) \& (\( \tau_0 = 0 \)) then
21: \( \text{return unschedulable} \)
22: if \( \Gamma^{ina} \neq 0 \) then
23: \( \text{return unschedulable} \)

**Algorithm 2** checkSchedulability(\( \theta, \pi \))

1: \( \text{schedulable} \leftarrow \text{true} \), \( \text{unschedulable} \leftarrow \text{false} \) \\
2: for all \( \theta_k \in \theta \) do \\
3: for all \( \tau_j \in \tau_k \) do \\
4: if \( \tau_j \) meets condition (13) then \\
5: \( \pi_k \text{.accept} (\tau_j) \) and \( \theta_k \text{.remove}(\tau_j) \) \\
6: for all \( \theta_k \in \theta \) do \\
7: if \( \theta_k \neq 0 \) then \\
8: \( \text{return unschedulable} \)
9: \( \text{return schedulable} \)
Complexity Analysis. We discuss the complexity in terms of the three stages. In the Start-up cache partitioning stage, we mainly execute groupbySensitivity and Start-upCachePartition. The first one enumerates m clusters for all tasks for maxl iterations. Hence, it takes \( O(n \cdot m \cdot \text{maxl}) \) time. Start-upCachePartition employs the MILP formulation, with in total \( 2m \) variables and \( m + 1 \) constraints. The complexity of this MILP problem is \( O(64^m \sqrt{m \ln m \ln m}) \) [11]. In the First-round cache partitioning stage, we assume that the allocated cache sets in this stage is \( s_1 \). It iteratively assigns the cache sets to dedicated cores and then conducting schedulability verification. This takes \( O(\frac{s_1 \cdot m \cdot n^2}{2}) \). In the Second-round cache partitioning stage, we assume allocated cache sets and tasks are \( s_2 \) and \( n_2 \) respectively. In every iterative cache allocation, we attempt to schedule the tasks from \( m - 1 \) cores. It takes \( O(\frac{s_2 \cdot m \cdot n_2}{2}) \). All in all, the complexity of the partitioned scheduling algorithm is \( O(64^m \sqrt{m \ln m \ln m}) + O(n \cdot m \cdot \text{maxl}) + O(\frac{s_1 \cdot m \cdot n^2 + s_2 \cdot m \cdot n_2}{2}) \). Despite the exponential complexity of the start-up stage, current MILP solver implementations are efficient and capable of solving real-world MILP formulations. Besides, for a specific taskset, we only apply the MILP formulation once in the initial cache partitioning phase at design time. We will demonstrate TCPS’ execution performance in Section 5.

5 Experimental Results

5.1 Experimental Set-up

From the Mälardalen [20] and TACLeBench [17] benchmarks, we select \( n \) tasks to generate 10,000 tasksets in each experiment. As for the taskset utilization \( U_{\text{tot}} \), we generate vectors consisting of \( n \) random elements using Randfixedsum [32], each representing an individual task utilization, and that are uniformly distributed in the designated value domain. Based on the generated utilization and realistic WCET, we can derive the \( T_r = \frac{T_r}{u_r} \). In our experiments, we measure the acceptance ratio for different taskset utilizations \( U_{\text{tot}} \), which is the number of schedulable tasksets divided by the total number of tasksets (i.e., 10,000), yielded by a scheduling method. The target platform is an ARMv7 processor with \( m \) identical cores, with private 4-way set-associative L1 data and instruction caches of 1KB each. The LLC is 8-way set-associative and 64KB in size. The access latencies of L1 caches, L2 cache and main memory are assumed to be 1, 10 and 100 cycles, respectively. For all partitioned cache approaches, we take those WCET values from the WCET curves that refer to the amount of cache an application actually has. For the shared cache approaches, we take the WCET values for a 1.0 normalized cache usage, i.e. the WCETs when applications use the entire cache.

5.2 Scheduling Policies

We compare TCPS with a large range of other non-preemptive scheduling approaches, covering all possible scheduling/cache partitioning combinations. To compare against global scheduling without cache partitioning, we study both EDF and Fixed Priority (FP) scheduling policies [38], referred to as GLB-edf and GLB-fp respectively. For global scheduling with cache partitioning, referred to as CP-edf, we use EDF scheduling and allocate an equal number of cache sets to each core. For partitioned scheduling with shared LLC, referred to as CITTA-1/C, we use the EDF-based algorithm from [39]. For schedulers using a shared LLC, we have extended the Heptane WCET estimation framework [21] with the modeling techniques from [38] to account for the cache interference in the schedulability analysis. In addition, we have adapted IA3 [29] by changing its coarse-grained hardware-based cache partitioning into fine-grained software-based partitioning, referred to as IA3*. Finally, to compare to the preemptive heuristic partitioned schedulers CATA [24] and CaM [40], we adjusted them such that they apply non-preemptive scheduling.

5.3 The Impact of Cache Sensitivity of Tasks on Schedulability

As explained in Section 3, we reconstruct monotonicity for counter-intuitive benchmarks using an approximation with upper and lower bound sensitivity curves. Figures 3b and 4a illustrate, for two experiments that will be elaborated below, the error introduced by this approximation. That is, the TCPS-l and TCPS_u curves in these figures represent the acceptance ratio using the lower and upper bound sensitivity curves, respectively. These results indicate that only around an additional 0.1% of tasksets are deemed schedulable only using lower bounds, i.e. the gap between these bounds is narrow.

To study the effects of cache sensitivity of tasks, we define cache-sensitive tasks (their WCET drops by 40% or more when receiving 12.5% of cache space compared to the minimum allocation space) and cache-insensitive tasks (their WCET drops by 10% or less when receiving 50% cache space). We randomly select \( n = 10 \) tasks to create 10,000 tasksets from these cache-sensitive and cache-insensitive workloads respectively to compare the acceptance ratio as shown in Figures 3a and 3b. We employ a t-test to compare the results for TCPS_u in Figure 3b and TCPS in Figure 3a. As the p-value of our test (0.0107) is less than alpha (0.05), we have sufficient evidence that TCPS schedules the cache-insensitive tasks more efficiently. This is due to the fact that cache space can be easily reduced for these tasks without substantial performance consequences, thereby saving cache space for more cache sensitive tasks in hybrid workloads. Moreover, the relative schedulability performance of TCPS compared to the
other scheduling methods is improving with lower cache sensitivity. E.g., the average schedulability gap between TCPS and IA^2+ is 15.15% and 10.89% in Figures 3b and 3a when \( U_{\text{tot}} \in [1.9, 2.9] \). The average schedulability gap between TCPS and CITTA-1/C is 23.64% and 9.23% in Figures 3b and 3a when \( U_{\text{tot}} \in \{1.5, 2.5\} \).

In the next experiment, we vary the degree of cache interference by using two types of tasksets with tasks of different code sizes. The mean and standard deviation of cache interference with respect to the entire execution time of tasks \( \tau_k \) in the high cache interference taskset is 6.45% and 13.11%, respectively. Likewise, the mean and standard deviation in the low cache interference taskset is 1.81% and 2.01%. Figures 4a and 4b show the acceptance ratio for these two types of tasksets. We observe that the schedulability performance of schedulers using a shared LLC (i.e., GLB-{edf,fp} and CITTA-1/C) significantly drops for heavy cache interference tasksets. CP-edf shows moderate schedulability performance, while TCPS keeps good performance and all tasksets can be distributed to cores successfully if \( U_{\text{tot}} \leq 2.9 \). Based on a t-test for the results in Figure 3, with a maximum p-value of 0.023 < alpha (0.05), we have sufficient evidence that TCPS outperforms IA^2+ and CATA in terms of schedulability. We note that IA^2+'s counter-intuitive schedulability instances in Figures 3b and 4a are due to the lack of reconstruction of WCET monotonicity. CaM exhibits similar high schedulability as TCPS when \( U_{\text{tot}} < 2.7 \), but for higher utilizations, CaM does not perform as well as TCPS. E.g., the schedulability of TCPS is 23.72% higher than with CaM on average when \( U_{\text{tot}} = 3.1 \) in Figure 3. This is because CaM randomly picks one permutation of clusters for mapping, additionally without considering the impact of carry-in jobs.

5.4 The Impact of Varying Number of Cores on Schedulability

We randomly select \( n = 20 \) tasks to generate 10,000 tasksets from the benchmarks while using 2, 4 or 8 cores as shown in Figure 5{a,b,c}. In Figure 5d, we randomly select \( n = 40 \) tasks to generate 10,000 tasksets from the benchmarks using 16 cores. For \( m = 2 \), GLB-{edf,fp} shows poor schedulability while TCPS exhibits relatively good performance (i.e., it outperforms all other approaches). However, when \( U_{\text{tot}} > 0.7 \),
TCPS also shows sub-optimal schedulability performance due to high DBF values caused by high core utilization and carry-in jobs. With the growing number of cores, GLB-{edf,fp} and CP-edf show limited improvement, whereas CITTA-1/C offers somewhat better scalability. In Figures 5b and 5c, the schedulability gap between CITTA-1/C and TCPS shrinks from 12.8% to 6.21% (with TCPS, m=6), respectively. TCPS (m=6) outperforms both CITTA-1/C and TCPS (m=8) because a growing number of cores implies less cache space per core. When \( m > 6 \), the cache space reduction per core, and resulting longer execution times, has a greater impact on schedulability than the decreased DBF and eliminated cache interference. We have also performed a t-test to compare CITTA-1/C and TCPS (m=8). Since the p-value (0.0022) is less than alpha (0.05), we can derive that TCPS outperforms CITTA-1/C. Figure 5c also indicates that the benefits of cache partitioning and partitioned scheduling in IA\(^3\) and CATA gradually cancel out since these heuristic algorithms (i.e., first-fit decreasing heuristic applied in IA\(^3\) and best-fit decreasing heuristic applied in CATA) do not perform holistic per-core cache allocation and task-to-core allocation.

Besides, they also do not explore the effect of carry-in jobs in the DBF function on scheduling performance.

We also observe from Figure 5d that the scalability of CITTA-1/C has its limits. This is because when increasing the number of cores that share the LLC, the growing WCET can potentially increase the upper bound on cache interference, eventually making the interfered tasks unschedulable. As shown in Figure 5d, TCPS and CaM achieve better schedulability performance at \( m=12 \). We performed a t-test to compare their schedulability performance. With a p-value of 0.00057, which is less than alpha (0.05), we can conclude that TCPS outperforms CaM (as well as all other approaches). This is due to the fact that TCPS employs the MILP formulation to find the optimal initial cache partitioning, which can accelerate further exploring of the cache partitioning solution space. Moreover, we take the carry-in jobs introduced by non-preemptive scheduling into account. Finally, we exploit task migrations to further improve the schedulability performance of the system, with a relatively good scalability (as demonstrated in Figure 5d).

Figure 5. The impact of a varying number of cores on schedulability.
5.5 Comparing Cache Usage and Load Balance

Figure 6 shows the cache usage for $U_{tot} \in \{1,2,3\}$ and load balance (i.e., the average variance of core utilization across $m$ cores) for those tasksets $\Gamma$ that are schedulable in Figure 5c and Figures 5[a,b,c,d] respectively. We observe that TCPS achieves more effective cache usage. E.g., it saves 10 cache sets compared with $IA^{3+}$, CaM and CATA when $U_{tot} = 1$. This is because TCPS employs the MILP to minimize cache usage. When $U_{tot} = 3$, the cache usage of TCPS and CATA are similar, but TCPS and $IA^{3+}$ can achieve better schedulability on 6 cores, saving 2 cores compared with CATA. Moreover, Figure 6 also shows that TCPS provides the most stable load balance, whereas $IA^{3+}$ suffers from heavy load unbalances since the second-round cache partitioning in TCPS tries to balance underutilized cores, especially for high $U_{tot}$.

5.6 Execution Time Analysis

We measure the execution time of the schedulability analysis for the eight scheduling policies in Figure 5b using 100 tasksets. Here, we included the partitioning overhead in the execution time for all evaluated partitioned methods (TCPS, CATA, $IA^{3+}$, CaM and CITTA). Figure 7 clearly shows that TCPS’ execution time is second only to CATA since CATA performs the cache partitioning without considering task characteristics. Despite the exponential complexity of TCPS’ start-up cache partitioning stage, the maximum running time is 74.39 seconds when $U_{tot}$ is 1.3, whereas the execution time of $IA^{3+}$ is 1.27x slower on average. This is probably due to the fact that $IA^{3+}$ employs a depth-first search for cache partitioning which explores plenty of invalid space compared with TCPS. The MILP solver to calculate the initial cache partitioning is a one-time job that can be efficiently addressed by the modern MILP libraries. The execution time of CITTA-1/C increases substantially as the utilization $U_{tot}$ increases and the average execution time is 4.21x slower than TCPS.

6 Related Work

**WCET estimation and cache interference in multicore.** In hard real-time systems, it is crucial to obtain the WCET
of each real-time task, which is used later to conduct schedu-
ability analysis. WCET estimation on a single-core has been
actively investigated in the past two decades. [36] provides
an excellent overview of WCET estimations. Unfortunately,
the existing techniques for uniprocessor architectures are not
applicable to multi-cores with shared caches. In our work, we
adopt cache partitioning to transform the multi-core schedul-
ing problem into a set of uniprocessor scheduling problems.
We exploit an extended version of Heptane [21, 39] to obtain
the WCET values. The extensions in Heptane, based on [37],
allow for calculating the cache interference between tasks
and integrating the upper bound WCET into the schedula-
bility analysis.

Cache partitioning. Cache partitioning is often sug-
gested to mitigate cache interference. There are two cache
partitioning methods: software-based [15, 35] and hardware-
based techniques [12, 19]. Page coloring [15, 35] is the most
common software-based cache partitioning schema, which
exploits the translation from virtual addresses to physical
addresses 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 tech-
niques such as cache locking mechanisms [27, 33, 34] and
Intel CAT [31] demand additional hardware component sup-
port which is not available in many commercial embedded
processors. In our implementation, we adopt software-based
cache partitioning.

Real-time scheduling. To schedule real-time tasks on
multi-core platforms, three paradigms are widely researched:
partitioned [7, 16, 18], global [4, 9, 25] and semi-partitioned
scheduling [10, 14, 23]. A comprehensive survey of real-time
scheduling for multi-core architecture can be found in [13].
Most of these research works assume that the WCETs are
estimated in an offline and isolated manner, i.e. the WCET
values are considered to be fixed.

[38] and [39] have explored real-time global and parti-
tioned scheduling for multi-core systems in the presence of
cache interference. Global scheduling with shared caches is
done via three steps: i) calculating the upper bound cache
interference given an execution window, ii) performing an
iterative algorithm to obtain the upper-bound cache interfer-
ence during task executions, and iii) integrating the upper
bound WCET into schedulability analysis. The authors later
extended this work to partitioned scheduling [39], called
CITTA, to distribute tasks to cores ahead to decrease the
cache interference. CITTA shows better schedulability com-
pared to global scheduling. However, CITTA still suffers
from cache interference across multi-cores. Moreover, a com-
prehensive comparison between real-time scheduling ap-
proaches using partitioned versus non-partitioned caches is
missing.

[24, 29, 40] distribute tasks to cores and then allocate cache
resources to cores based on preemptive scheduling. How-
ever, these works do not explicitly calculate the preemption
overhead during task execution. In addition, the bandwidth-
aware partitioning [40] is based on a soft real-time system.
The above limitations result in insufficient guarantees for
hard real-time requirements. In our work, we calculate the
WCET by Heptane which has been extended to calculate the
cache interference among tasks which provides the safety
and predictability for hard real-time systems.

7 Conclusion

Shared LLC caches in multi-core processors introduce se-
rious difficulties in providing guarantees on the real-time
properties of embedded software due to the interaction and
the resulting contention in the shared caches. Cache par-
titioning is often suggested as a means of mitigating the
intra-core interference. However, the constrained cache us-
age may potentially increase the WCET which also degrade
the schedulability performance.

This paper presents a non-preemptive partitioned sched-
uling algorithm called TCPS that aims at improving system
schedulability through awareness of the cache sensitivity
characteristics of applications. TCPS combines partitioned
scheduling with cache partitioning to mitigate the cache in-
terference problem in real-time multi-core systems. As a first
step, TCPS clusters tasks based on their WCET sensitivity
to cache partition size and associates each of these clusters
with a different core. Then, a MILP problem is formulated to
get the initial cache partition size for each core. A number of
refinement steps are then carried out to refine the initial allo-
cation and potentially extend the initial cache partition sizes.
Once the task-to-core allocation is completed, partitioned
EDF is employed online.

We have compared the schedulability performance of
TCPS with a large range of state-of-the-art approaches, cov-
ering all possible scheduling/cache partitioning combina-
tions. Our evaluation includes schedulability experiments
using different task characteristics (e.g., cache sensitive vs.
cache insensitive tasks), and a varying number of deployed
cores. Besides schedulability performance, we also studied
the execution times of schedulers as well as their cache effi-
ciency and load balance. Our empirical evaluation shows that
TPCS outperforms all other studied scheduling approaches
in terms of schedulability performance while also yielding a
more effective cache usage and more stable load balancing.

TCPS exploits the task characteristics and their cache de-
mand for partitioning. However, in this work, we ignore
any thermal effects. For instance, some similar compute-
-intensive tasks grouped as a cluster may contribute to a
thermal hotspot, resulting in a lower performance and thus
impacting the schedulability of the system. In our future
work, we plan to employ our scheduler in the HotSniper[30]
simulator, which accounts for the system’s thermal behav-
ior, to try to realize a more robust scheduler for real-time
systems.
References


[18] 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. Citeseer. https://doi.org/10.1109/TC.2006.113


[31] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and
Maria E Gómez. 2017. Application clustering policies to address system
fairness with intel’s cache allocation technology. In 2017 26th interna-
tional conference on parallel architectures and compilation techniques
(pact). IEEE, 194–205. https://doi.org/10.1109/PACT.2017.19

2014. Efficient data generation for the testing of real-time multi-
https://doi.org/10.12915/pe.2014.09.36

2012. Semi-partitioned hard-real-time scheduling under locked cache
migration in multicore systems. In 2012 24th Euromicro Conference
on Real-Time Systems. IEEE, 331–340. https://doi.org/10.1109/ECRTS.
2012.27

[34] Vivy Suhendra and Tulika Mitra. 2008. Exploring locking & parti-
tioning for predictable shared caches on multi-cores. In Proceed-
ings of the 45th annual Design Automation Conference. 300–303.
https://doi.org/10.1145/1391469.1391545

[35] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H
Anderson. 2013. Outstanding paper award: Making shared caches more
predictable on multicore platforms. In 2013 25th Euromicro Conference
2013.26

[36] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti,
Stephan Thesing, David Whalley, Guillem Bernard, Christian Ferdi-
execution-time problem—overview of methods and survey of tools.
ACM Transactions on Embedded Computing Systems (TECS) 7, 3 (2008),
1–53. https://doi.org/10.1145/1347375.1347389

[37] Jun Xiao, Sebastian Altmeyer, and Andy Pimentel. 2017. Schedulabil-
ity Analysis of Non-preemptive Real-Time Scheduling for Multicore
Processors with Shared Caches. In 2017 IEEE Real-Time Systems Sym-
posium (RTSS). 199–208. https://doi.org/10.1109/RTSS.2017.00026

[38] Jun Xiao, Sebastian Altmeyer, and Andy D. Pimentel. 2020. Schedu-
lability Analysis of Global Scheduling for Multicore Systems With
//doi.org/10.1109/TC.2020.2974224

aware Task Partitioning for Non-preemptive Real-time Multi-core
Systems. ACM Transactions on Embedded Computing Systems (TECS)

[40] Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, Yuhan Lin, Haoran
Li, Chenyang Lu, and Insup Lee. 2019. Holistic resource allocation
for multicore real-time systems. In 2019 IEEE Real-Time and Embedded
Technology and Applications Symposium (RTAS). IEEE, 345–356. https:
//doi.org/10.1109/RTAS.2019.00036