Schedulability Analysis of Global Scheduling for Multicore Systems With Shared Caches

Xiao, J.; Altmeyer, S.; Pimentel, A.D.

DOI
10.1109/TC.2020.2974224

Publication date
2020

Document Version
Final published version

Published in
IEEE Transactions on Computers

License
Article 25fa Dutch Copyright Act

Citation for published version (APA):
Schedulability Analysis of Global Scheduling for Multicore Systems With Shared Caches

Jun Xiao †, Member, IEEE, Sebastian Altmeyer, Member, IEEE, and Andy D. Pimentel ‡, Senior Member, IEEE

Abstract—Shared caches in multicore 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. To address this problem, we develop a new schedulability analysis for real-time multicore systems with shared caches, globally scheduled by Earliest Deadline First (EDF) and Fixed Priority (FP) algorithms. We construct an integer programming formulation, which can be transformed to an integer linear programming formulation, to calculate an upper bound on cache interference exhibited by a task within a given execution window. Using the integer programming formulation, an iterative algorithm is presented to obtain the upper bound on cache interference a task may exhibit during one job execution. The upper bound on cache interference is subsequently integrated into the schedulability analysis to derive a new schedulability condition. A range of experiments is performed to investigate how the schedulability is degraded by shared cache interference. We also evaluate the schedulability performance of EDF against FP scheduling over randomly generated tasksets. Our empirical evaluations show that EDF is better than FP scheduling in terms of the number of task sets deemed schedulable.

Index Terms—Real-time systems, multi-core systems, schedulability analysis, shared caches, global scheduling

1 INTRODUCTION

MULTICORE architectures are increasingly used in both the desktop and the embedded markets. Modern multicore processors incorporate shared resources between cores to improve performance and efficiency. Shared caches are among the most critical shared resources on multicore systems as they can efficiently bridge the performance gap between memory and processor speeds by backing up small private caches. 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.

In a multicore processor with shared caches, a real-time task may suffer from two different kinds of cache interferences [1], which severely degrade the timing predictability of multicore systems. The first is called intra-core cache interference, which occurs within a core, when a task is preempted and its data is evicted from the cache by other real-time tasks. The second is inter-core cache interference, which happens when tasks executing on different cores access the shared cache simultaneously. Inter-core cache interference may cause several types of cache misses including capacity misses, conflict misses and so on [2]. 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 challenging to design real-time applications executing on multicore platforms with shared caches, which 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. With a multicore system, the WCETs are strongly dependent on the amount of inter-core interference on shared hardware resources such as main memory, shared caches and interconnects. In this paper, we shall only focus on the shared cache interferences and study the schedulability analysis problem for hard real-time tasks that exhibit shared cache interferences.

A major obstacle is to predict the cache behavior to accurately obtain the WCET of a real-time task considering inter-core cache interference since different cache behaviors (cache hit or miss) will result in different execution times of each instruction. In [3], it was even pointed out that “it will be extremely difficult, if not impossible, to develop analysis methods that can accurately capture the contention among multiple cores in a shared cache”. In this paper, we assume that a task’s WCET itself does not account for shared cache interference but, instead, we determine this interference explicitly (as will be explained later on). [4] presents such an approach to derive a task’s WCET without considering shared cache interference.

This paper proposes a novel schedulability analysis of global real-time scheduling for multicore systems with shared caches. We construct an integer programming formulation, which can be transformed to an integer linear programming formulation, to calculate an upper bound on cache interference exhibited by a task within a given execution window. Using the integer programming formulation, an iterative
algorithm is presented to obtain the upper bound on cache interference a task may exhibit during one job execution. The upper bound on cache interference is subsequently integrated into the schedulability analysis to derive a new schedulability condition. A range of experiments is performed to investigate how the schedulability is degraded by shared cache interference for a range of different tasksets.

The original version of our schedulability analysis for real-time multicore systems with shared caches was presented in [5]. Significant extensions are made in this paper, including:

- A more general framework for the schedulability analysis of global scheduling, accounting for shared cache interference. The original scheduling analysis mainly focuses on FP scheduling, while the extended scheduling analysis presented in this work applies not only to FP scheduling but also to EDF scheduling;
- Evaluation of the schedulability performance of EDF against FP scheduling over randomly generated tasksets. Our empirical evaluations show that EDF is slightly better than FP scheduling in terms of task sets deemed schedulable.

The rest of the paper is organized as follows. Section 2 gives an overview of the related work. The system model is described in Section 3. Section 4 describes the proposed schedulability analysis, where we also detail the computation of processor-contention and inter-core cache interferences applied in the analysis. Section 5 presents an iterative computation to obtain the upper bound of inter-core cache interferences. Section 6 presents the experimental results, after which Section 7 concludes the paper.

2 RELATED WORK

WCET Estimation. For hard real-time systems, it is essential to obtain each real-time task’s WCET, which provides the basis for the schedulability analysis. WCET analysis has been actively investigated in the last two decades, of which an excellent overview can be found in [6]. There are well-developed techniques to estimate real-time tasks’ WCET for single processor systems. Unfortunately, the existing techniques for single processor platforms are not applicable to multicore systems. Only a few methods have been developed to estimate task WCETs for multicore systems with shared caches [7], [8], [9]. In almost all those works, due to the assumption that cache interferences can occur at any program point, WCET analysis will be extremely pessimistic, especially when the system contains many cores and tasks. An overestimated WCET is not useful as it degrades system schedulability.

Shared Cache Interference. Since shared caches considerably complicate the task of accurately estimating the WCET, many researchers in the real-time systems community have recognized and studied the problem of cache interference in order to use shared caches in a predictable manner. Cache partitioning, which isolates application workloads that interfere with each other by assigning separate shared cache partitions to individual tasks, is a successful and widely-used approach to address contention for shared caches in (real-time) multicore applications. There are two cache partitioning methods: software-based and hardware-based techniques [10]. The most common software-based cache partitioning technique is page coloring [11], [12], [13], [14]. By exploiting the virtual to physical page address translations present in virtual memory systems at OS-level, page addresses are mapped to predefined cache regions to avoid the overlap of cache spaces. While cache partitioning technique using page coloring has the following drawbacks. First, it requires heavy modifications to virtual memory subsystem in the operating system. Second, the number of partitions is limited as a cache partition is coarsely sized (multiples of page size × cache ways). Hardware-based cache partitioning is achieved using a cache locking mechanism [3], [13], [15], which prevents cache lines from being evicted during program execution. For example, [16] presented vCAT, an approach for dynamic shared cache management on multicore virtualization platforms based on Intel’s Cache Allocation Technology (CAT). The drawback of cache locking is that it requires specific hardware support that is not available in many commercial processors. Cache way-partitioning like CAT has also significant limitation due to a small number of coarsely-sized partitions (in multiples of way size).

Real-Time Scheduling. The schedulability analysis of global multiprocessor scheduling has been intensively studied [17], [18], [19], [20], [21], [22], [23], of which comprehensive surveys can be found in [24], [25]. Most multi-core scheduling approaches assume that the WCETs are estimated in an offline and isolated manner and that WCET values are fixed.

A few works address schedulability analysis for multi-core systems with shared caches [26], [27], [28], but these works deployed cache partitioning techniques. 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 [28].

Different from the above work, we developed a new schedulability analysis of global scheduling for multicore systems in which cache space isolation techniques are not deployed. Instead of using cache partitioning to eliminate shared cache interference, we focus on the analysis of shared cache interference that a task may exhibited during its execution. Our approach neither requires operating system modifications for page coloring nor hardware features for cache locking.

Our work also differs from other approaches to the timing verification of multicore systems [29] in that all other sources of interferences are assumed to be included within the WCET. We analyze the effect of shared cache interference on the schedulability. To the best of our knowledge, this is among the first works that integrates inter-core cache interference into schedulability analysis.

3 SYSTEM MODEL

3.1 Task Model

We consider a set \( \tau \) of periodic or sporadic real-time tasks \( \{t_1, t_2, \ldots, t_n\} \) to be scheduled on a multicore processor. Each task \( t_k = (C_k, D_k, T_k) \in \tau \) 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. Moreover, jobs of any task cannot be executed at the same time on more than one core. A task \( t_k \) is a sequence of jobs \( j_l \), where \( j \) is the job index. We denote the arrival time, starting time, finishing time and absolute deadline of a job \( j \) as \( r_l \), \( s_l \), \( f_l \) and \( d_l \), respectively. Note that the goal of a real-time scheduling algorithm is to guarantee that each job will complete before its absolute deadline: \( f_l \leq d_l = r_l + D_k \).

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

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

### 3.2 Architecture Model

Our system architecture consists of a multicore processor with \( m \) identical cores onto which the individual tasks are scheduled. Most multicore processors have instruction and data caches. Caches are organized as a hierarchy of multiple cache levels to address the tradeoff between cache latency and hit rate. The low level caches (L1) in our considered multicore processor are assumed to be private, while the last level caches (LLC, for example L2) are shared between all cores. Furthermore, we assume that the LLC cache is non-inclusive with respect to the private caches (L1), and that LLC caches are direct-mapped caches.

Data caches, in general, are hard to analyze statically. In this work, we focus on instruction caches and we adopt the approach in [4] to derive task WCET. The analysis would require further extension in order to be applied to data caches.

### 3.3 Global Schedulers

In this paper, we focus on non-preemptive global scheduling. Once a task instance starts execution, any preemption during the execution is not allowed, so it must run to completion. So we do not have to consider intra-core cache interference. If not explicitly stated, cache interference will therefore refer to inter-core cache interference in the following discussion. We consider two well-known global scheduling algorithms: Non-Preemptive Earliest Deadline First (EDF<sub>np</sub>) and Non-Preemptive Fixed Priority (FP<sub>np</sub>).

EDF<sub>np</sub> 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. Since each job’s absolute deadline changes over time, the priority of a task changes dynamically.

For FP<sub>np</sub> scheduling, a fixed priority \( P_k \) is assigned to each task \( t_k \) \((k = 1, 2, \ldots, n)\). As each task has a unique priority, we use \( h_p(k) \) to denote the set of tasks with higher priorities than \( t_k \), and \( h_c(p)(k) = h_p(k) \cup \{ t_k \} \) the set of tasks whose priorities are not lower than \( t_k \). Similarly, \( l_p(k) \) is the set of tasks with lower priorities than \( t_k \) and \( l_c(p)(k) = l_p(k) \cup \{ t_k \} \) the set of tasks whose priorities are not higher than \( t_k \).

The EDF<sub>np</sub> and FP<sub>np</sub> scheduling algorithms are work-conserving, according to the following definition.

**Definition 1.** A scheduling algorithm is work-conserving if there are no idle cores when a ready task is waiting for execution.

### 4 Schedulability Analysis

In this section, we give an overview of the new schedulability analysis that accounts for cache interference. We also present the approaches to derive the upper bound on the parameters used in the schedulability condition.

#### 4.1 Overview

We first analyze the execution of one job \( J_k \) of a task \( t_k \). Let \( s_k \) denote the latest time-instant no later than \( r_k \) \((o_k' \leq r_k')\) at which at least one processor is idle and let \( A_k = r_k - o_k' \). As all processors are idle when the system starts, there always exists such a \( o_k' \). The time interval \([o_k', d_k']\) can be divided into two parts \([o_k', s_k']\) and \([s_k', d_k']\).

As shown in Fig. 1, a job \( J_k \) of task \( t_k \) exhibits two kinds of interferences during \([o_k', s_k']\). The first interference is called processor-contention interference, denoted by \( I^c_k \). It is the cumulative length of all intervals over \([o_k', s_k']\) in which all the processing cores are busy executing jobs other than \( J_k \). We define the interference \( I^c_k \) of a task \( t_k \) on a task \( t_k \) over the interval \([o_k', s_k']\) as the cumulative length of all intervals in which \( t_k \) is executing. The second type of interference is the cumulative length of all extra execution delays caused by shared cache interference from all other tasks running concurrently on other cores, denoted as \( I^c_k \). We also define the interference \( I^c_k \) as the cumulative length of all extra execution delays of \( t_k \) caused by shared cache accesses between task \( t_k \) and task \( t_k \).

Furthermore, we define the upper bound on processor-contention interference as \( I^c_k \) and similarly the upper bound on shared cache interference as \( I^c_k \).

Note that the processor-contention interference \( I^c_k \) occurs during \([o_k', s_k']\), so \( I^c_k \) depends on \( A_k \) and the length of \([r_k, s_k']\). While the shared cached interference \( I^c_k \) occurs only during \( t_k \)’s execution. We will present the derivation of \( I^c_k \) in the next section and it can be shown that \( I^c_k \) does not depend
on $A_k$ and the length of $[r_k^i, s_k^i]$. Let us now assume $I_k^{sc}$ is known.

We can compute the latest start time of job $J_k^i$ from task $r_k: l_k^i = d_k^i - C_k - I_k^{sc}$, i.e., if $J_k^i$ starts its execution before $l_k^i$, it will be able to finish execution before deadline $d_k^i$. The length of $[r_k^i, l_k^i]$ is $S_k = D_k - C_k - I_k^{sc}$. Since we consider non-preemptive scheduling, in order for $J_k^i$ to miss its deadline, all $m$ cores must be continuously busy executing tasks other than $r_k$ in the time interval $[o_k^i, l_k^i]$. In other words, if $S_k < 0$, $J_k^i$ will miss its deadline. Therefore, we name the time interval $[o_k^i, l_k^i]$ as a problem window. We assume $S_k \geq 0$ in the following description.

As the proccessor-contention interference only occurs before the start of the $r_k$'s execution, we restrict $I_k^{pre}$, $I_k^{pre}$ and $I_k^{pre}$ to the time interval $[o_k^i, l_k^i]$.

By construction, we have the first schedulability test for $r$.

**Theorem 1.** A task set $\tau$ is schedulable with an $EDF_{np}$ or $FP_{np}$ scheduling policy on a multicore processor composed of $m$ identical cores with shared caches if for each task $r_k \in \tau$ and all $A_k \geq 0$

$$I_k^{pre} + C_k + I_k^{sc} < D_k + A_k.$$ 

### 4.2 Computation of $I_k^{pre}$

The workload $W_{i,k}$ of a task $r_k$ is the time task $r_k$ executes during time interval $[o_k^i, l_k^i]$ of length $A_k + S_k$, according to a given scheduling policy.

**Lemma 1.** The processor-contention interference that a task $r_k$ causes on a task $r_i$ in $[o_k^i, l_k^i]$ is never greater than the workload of $r_k$ in $[o_k^i, l_k^i]$

$$\forall i, k, j \ I_k^{pre} \leq W_{i,k}.$$ 

Lemma 1 is obvious, since $W_{i,k}$ is an upper bound on the execution of $r_k$ in $[o_k^i, l_k^i]$.

Note that $r_k$ may execute more than $C_k$ due to the shared cache interference. That is, the actual execution time of $r_k$'s job is bounded by $C_k^r = C_k + I_k^{sc}$. In the following discussion, we use $C_k^r$ as the upper bound on the workload contribution from a single job of $r_k$.

As the number of $r_k$'s jobs released in $[o_k^i, l_k^i]$ is at most $\left\lfloor \frac{A_k + S_k}{T_k^r} \right\rfloor$, $W_{i,k}$ can be roughly bounded by $\frac{A_k + S_k}{T_k^r} \times C_k^r$. However, a tighter upper bound on the worst-case workload can be calculated by categorizing each job of $r_k$ in $[o_k^i, l_k^i]$ into one of the three types [30]:

- carry-in job: a job with its release time earlier than $o_k^i$ but with its deadline earlier than $l_k^i$;
- body job: a job with both its release time and its deadline in $[o_k^i, l_k^i]$;
- carry-out job: a job with its release time in $[o_k^i, l_k^i]$, but with its deadline later than $l_k^i$.

As shown in Fig. 2, the worst-case workload of $r_k$ occurs when a carry-in job (if $r_k$ has a carry-in job) finishes execution as late as possible and a carry-out job starts its execution as early as possible. We use $W_{i,k}^{\alpha}$ to denote an upper bound of $r_k$'s workload in $[o_k^i, l_k^i]$ if $r_k$ has no carry-in job, and use $W_{i,k}^{\gamma}$ to denote an upper bound of $r_k$'s workload if $r_k$ has a carry-in job.

Following the approach in [19], we derive a tighter upper bound on $W_{i,k}^{\alpha}$ and $W_{i,k}^{\gamma}$ for the $EDF_{np}$ and $FP_{np}$ scheduling policies, separately. We omit the proof due to space limitations. Interested readers can refer to [19] for a detailed explanation.

#### 4.2.1 Upper Bound on $W_{i,k}^{\alpha}$ for $EDF_{np}$

$EDF_{np}$ assigns a priority of a job by the absolute deadline of that job. We have the following lemma.

**Lemma 2.** For $EDF_{np}$, if $D_i > D_k$, the necessary condition for $J_k^i$ to cause interference to $J_k^j$ is $r_k^i < r_k^j$, i.e., $J_k^i$ must be released earlier than $J_k^j$; if $D_i \leq D_k$, the necessary condition for $J_k^i$ to cause interference to $J_k^j$ is $d_i \leq d_k$, i.e., $J_k^i$'s absolute deadline must be no later than that of $J_k^j$.

**Proof.** Lemma 2 is from [19]. See the proof of Lemma 2 in [19].

Since $r_k$ has no carry-in jobs in this case, the worst case of $W_{i,k}^{\alpha}$ occurs when the first job of $r_k$ is released at time $o_k^i$. The next jobs of $r_k$ are then released periodically every $T_k$ time units. Thus, $W_{i,k}^{\alpha}$ is computed by three cases: (1) $i = k$, (2) $D_i \leq D_k$, (3) $D_i > D_k$.

1. $i = k$. As shown in Fig. 3, only body jobs in $[o_k^i, r_k^i]$ contribute to processor-contention interference and the number of $r_k$'s body instances is $\left\lfloor \frac{A_k}{T_k} \right\rfloor$. So we have

$$W_{i,k}^{\alpha} = \left\lfloor \frac{A_k}{T_k} \right\rfloor C_k^r.$$ 

2. $D_i \leq D_k$. Fig. 4 shows the worst case of $W_{i,k}^{\alpha}$ for $D_i \leq D_k$. The number of body jobs of $r_i$ is $\left\lfloor \frac{A_k + S_k}{T_k} \right\rfloor$.

![Fig. 3. The densest possible packing of jobs of $r_i$ without carry-in job, if $i = k$.](image-url)

![Fig. 4. The densest possible packing of jobs of $r_i$ without carry-in job and $D_i \leq D_k$. Case (a): $\alpha \leq A_k + D_k$, Case (b): $\alpha > A_k + D_k$.](image-url)
We use \( \alpha \) to denote the distance between \( o^i_l \) and the deadline of \( \tau_i \)'s carry-out job, \( \alpha = \frac{A_k + S_k}{T_i} T_i + D_i \). The deadline of \( \tau_i \)'s carry-out job is \( o^i_l + \alpha \).

(2.A) If \( \alpha \leq A_k + D_k \), as shown in case (a) in Fig. 4, the contribution of the carry-out job is bounded by \( \min(C^*_i, (A_k + S_k) \mod T_i) \). In this case, we have

\[
W^n_{i,k} = \frac{A_k + S_k}{T_i} C^*_i + \min(C^*_i, (A_k + S_k) \mod T_i). \tag{2}
\]

(2.B) If \( \alpha > A_k + D_k \), shown as case (b) in Fig. 4, the contribution of the carry-out job is 0, we have

\[
W^n_{i,k} = \frac{A_k + S_k}{T_i} C^*_i. \tag{3}
\]

3) \( D_i > D_k \). Fig. 5 shows the worst case of \( W^n_{i,k} \) for \( D_i > D_k \). The number of body jobs of \( \tau_i \) is \( \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor \).

By Lemma 2, a job of \( \tau_i \) can interfere with \( J^l_k \) only if its release time is earlier than \( r^i_l \). We use \( \beta \) to denote the distance between \( o^i_l \) and the release time of \( \tau_i \)'s carry-out job, \( \beta = \frac{A_k + S_k}{T_i} T_i \).

(3.A) If \( A_k = 0 \), then \( o^i_l = r^i_l \). Since \( D_i > D_k \), any task instance released no earlier than \( o^i_l \) has a deadline later than \( d^i_l \), so, \( W^n_{i,k} = 0 \).

(3.B) If \( \beta < A_k \), shown as case (a) in Fig. 5. The contribution of \( \tau_i \)'s carry-out job is bounded by \( \min(C^*_i, (A_k + S_k) \mod T_i) \). \( W^n_{i,k} \) is computed by Equation (2).

(3.C) If \( \beta > A_k > 0 \), as shown in Fig. 5 case (b), the contribution of \( \tau_i \)'s carry-out job is 0, and \( W^n_{i,k} \) is computed by Equation (3).

By the discussions above, we can compute \( W^n_{i,k} \) for \( EDF_{np} \) by

\[
W^n_{i,k} = \begin{cases} 
0 & D_i > D_k \land A_k = 0 \\
W^{n^1}_{i,k} & i = k \\
W^{n^2}_{i,k} & (i \neq k \land D_i \leq D_k \land A \leq A_k + D_k), \\
& \lor (D_i > D_k \land \beta < A_k) \\
W^{n^3}_{i,k} & \text{otherwise}
\end{cases} \tag{4}
\]

where \( W^{n^1}_{i,k}, W^{n^2}_{i,k}, W^{n^3}_{i,k} \) are defined in Equations (1), (2) and (3) respectively.

4.2.2 Upper Bound on \( W^n_{i,k} \) for \( EDF_{np} \)

We now compute the upper bound on \( W^n_{i,k} \) by four cases: (1) \( i = k \), (2) \( D_i \leq D_k \) and \( S_i > C^*_i \), (3) \( D_i > D_k \) and \( S_i \geq C^*_i \).

1) \( i = k \), shown in Fig. 6. The number of body jobs of \( \tau_k \) is \( \left\lfloor \frac{A_k}{T_k} \right\rfloor \). The contribution of the carry-in job is bounded by \( \min(C^*_i, \max(0, (A_k \mod T_k) - T_k + D_k)) \). So in this case, we have

\[
W^n_{i,k} = \frac{A_k}{T_k} C^*_i + \min(C^*_i, \max(0, (A_k \mod T_k) - T_k + D_k)). \tag{5}
\]

2) \( D_i \leq D_k \land S_i > C^*_i \). Shown as case (a) in Fig. 7, the worst case of \( W^n_{i,k} \) occurs when \( \tau_i \)'s last released instance has its deadline at \( d^i_l \). The number of \( \tau_i \)'s body jobs is \( \left\lfloor \frac{A_k + D_k}{T_i} \right\rfloor \). The contribution of the carry-in job is bounded by \( \min(C^*_i, (A_k + D_k) \mod T_i) \). So, we have

\[
W^n_{i,k} = \frac{A_k + D_k}{T_i} C^*_i + \min(C^*_i, (A_k + D_k) \mod T_i). \tag{6}
\]

3) \( D_i > D_k \land S_i \geq C^*_i \). Case (b) in Fig. 7 shows the worst case of \( W^n_{i,k} \). By Lemma 2, \( \tau_i \)'s job can interfere with \( J^l_k \) only if its release time is earlier than \( r^i_l \). So, the worst case of \( W^n_{i,k} \) occurs when one of \( \tau_i \)'s instances is released at \( r^i_l - 1 \).

(3.A) If \( A_k > 0 \), the number of \( \tau_i \)'s body instances is \( \left\lfloor \frac{A_k - 1}{T_i} \right\rfloor \), the carry-out is \( C^*_i \), the carry-in is bounded by \( \mu = \min(C^*_i, \max(0, (A_k - 1) \mod T_i - (T_i - D_k))) \).

(3.B) If \( A_k = 0 \), only the carry-out job contributes at most \( C^*_i - 1 \). So, we have

\[
W^n_{i,k} = \begin{cases} 
C^*_i - 1 & A_k = 0 \\
\left( \left\lfloor \frac{A_k - 1}{T_i} \right\rfloor + 1 \right) C^*_i + \mu & A_k > 0
\end{cases} \tag{7}
\]

4) For the remaining cases, i.e., \( (D_i \leq D_k \land S_i \geq C^*_i) \lor (D_i > D_k \land S_i < C^*_i) \), the worst case of \( W^n_{i,k} \) occurs...
when one of \( r_i \)'s instances is released at \( T_k \) as shown in Fig. 8.

(4.A) If \( A_k + S_k \leq C_i \), then \( W_{i,k}^c = A_k + S_k \).

(4.B) If \( A_k + S_k > C_i \), the number of \( r_i \)'s body job is \( \left\lfloor \frac{A_k + S_k - C_i}{T_i} \right\rfloor \), the contribution of the carry-out job is \( C_i \); carry-in is bounded by \( v = \min(C_i, \max(0, (A_k + S_k - C_i) \mod T_i - (T_i - D_i))) \).

\[
W_{i,k}^c = \begin{cases} 
A_k + S_k & A_k + S_k \leq C_i \\
\left\lfloor \frac{(A_k + S_k - C_i)}{T_i} \right\rfloor + 1 & A_k + S_k > C_i 
\end{cases}
\]  

(8)

By the discussion above, we compute \( W_{i,k}^c \) for EDF\(_{np} \) by

\[
W_{i,k}^c = \begin{cases} 
W_{i,k}^{c_1} & i = k \\
W_{i,k}^{c_2} & i \neq k \land D_i \leq D_k \land S_k > C_i \\
W_{i,k}^{c_3} & D_i > D_k \land S_k \geq C_i \\
W_{i,k}^{c_4} & \text{otherwise}
\end{cases}
\]  

(9)

where \( W_{i,k}^{c_1}, W_{i,k}^{c_2}, W_{i,k}^{c_3} \) and \( W_{i,k}^{c_4} \) are defined in Equations (5), (6), (7) and (8) respectively.

### 4.2.3 Upper Bound on \( W_{i,j}^c \) for FP\(_{np} \)

The following lemma describes the condition of processor-contention interference on \( r_i \) caused by lower-priority tasks in \( lp(k) \) for FP\(_{np} \).

**Lemma 3.** For FP\(_{np} \), a task instance \( J_i^f \) of \( r_i \in lp(k) \) can interfere with \( J_j^f \) only if \( J_j^f \) is released before \( r_k^f \).

We compute the upper bound on \( W_{i,j}^c \) by three cases: (1) \( i = k \), (2) \( \tau_i \in lp(k) \), (3) \( \tau_i \in lp(k) \).

1) \( i = k \). The worst-case workload is the same as in the case of EDF\(_{np} \) thus \( W_{i,k}^c \) can be computed by Equation (1).

2) \( \tau_i \in lp(k) \). The worst-case workload of task \( \tau_i \) occurs when a job of \( \tau_i \) arrives at \( o_i \) as shown in case (a) in Fig. 4. \( W_{i,k}^c \) can be computed using Equation (2).

3) \( \tau_i \in lp(k) \). The worse case of \( W_{i,k}^c \) occurs when one of \( \tau_i \)'s instances is released at \( o_i^j \). The number of body jobs of \( \tau_i \) is \( \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor \). Let \( y \) be the distance between \( o_i^j \) and the release time of \( \tau_i \)'s last instance. So \( y = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor \).

(3.A) If \( A_k = 0 \), then \( o_i^j = r_i^f \). According to Lemma 3, \( W_{i,k}^c = 0 \).

(3.B) If \( y < A_k \), \( \tau_i \)'s last job is released earlier than \( r_i^f \) as shown in Fig. 9 case (a), its contribution is bounded by \( \min(A_k + S_k \mod T_i, C_i) \). In this case, \( W_{i,k}^c \) is computed by Equation (2).
where \( r^a \) and \( r^c \) satisfy \( r^a \cup r^c = \tau \), \( r^a \cap r^c = \emptyset \) and \( |r^c| \leq m - 1 \).

By taking the maximum over the task set, \( \Omega_k \) describes an upper bound on the total worst-case workload in \( [\sigma_k, \epsilon_k] \). The complexity to compute \( \Omega_k \) is \( O(n) \), as explained in [18].

Since both \( EDF_{np} \) and \( FP_{np} \) are work-conserving, the processor-contention interference exhibited by \( \tau_k \) can be bounded by \( \frac{\Omega_k}{m} \). So, we have the following Lemma.

**Lemma 4.** If tasks are scheduled with an \( EDF_{np} \) or \( FP_{np} \) scheduling policy on a multicore processor composed of \( m \) identical cores with shared cache

\[
I_k^{pre} \leq \frac{\Omega_k}{m}.
\]

The pessimism of the analysis of upper bound on the processor-contention interference mainly comes from the assumption that every tasks take the their worst-case execution time and the computational loads are equally distributed to \( m \) cores.

### 4.3 Computation of \( \bar{I}_k^{rec} \)

We first identify the maximum cache interference between two tasks and then we construct an integer programming formulation to calculate the upper bound on the shared cache interference exhibited by a task within an execution window.

#### 4.3.1 Cache Interference Between Two Tasks

We first analyze the cache interference during one job execution between \( \tau_k \) and \( \tau_t \). Let \( \tau_k \) be the interfered and \( \tau_t \) be the interfering task.

Following the approach in [4], we can obtain the WCET of a task by performing a Cache Access Classification (CAC) and Cache Hit/Miss Classification (CHMC) analysis for each instruction memory access at the private caches and the shared LLC cache separately.

**CAC and CHMC.** The CAC determines the possibility that an instruction being fetched from memory will access a certain cache level, and the access to a certain cache level can be Always (A), Uncertain (U) or Never (N). A reference \( r \) at a cache level \( L \) is considered as \( A \) if the access to \( r \) is always performed at cache level \( L \) and \( r \) is considered as \( N \) if the access to \( r \) is never performed at cache level \( L \), while the access is classified as \( U \) if it is not \( A \) nor \( N \). CHMC assigns a cache lookup result to each memory reference according to the cache states. As a result, a reference to a memory block of instructions can be classified as Always Hit (AH), Always Miss (AM) or Uncertain (U).

The CAC for a reference \( r \) at a cache level \( L \) depends on the results of CAC and CHMC of the reference \( r \) at the level \( L-1 \). Since we consider noninclusive caches, accesses to the private caches cannot be affected by tasks executing on other cores. Accesses classified as \( AM \) or \( U \) at the shared LLC cache will also not be affected by shared cache interferences, since they are already counted as misses in the WCET analysis.

We start the cache interference analysis by defining two concepts for cache blocks.

**Definition 2.** A Hit Block (HB) is a memory block whose access is classified as \( AH \) at the shared LLC cache.

**Definition 3.** A Conflicting Block (CB) is a memory block whose access is classified as \( A \) or \( U \) at the shared LLC cache.

\( HB \) and \( CB \) can be identified by the approach proposed in [4].

We use \( HB_k = \{m_{k,1}, m_{k,2}, \ldots, m_{k,p}\} \) to represent the set of \( HB \) for task \( \tau_k \) and use \( n_{k,x} (x = 1, 2, \ldots, p) \) to denote the number of \( m_{k,x} \)'s accesses that are classified as an \( AH \) at the LLC cache. Similarly, we define \( CB_i = \{m_{i,1}, m_{i,2}, \ldots, m_{i,q}\} \) as the set of \( CB \) for task \( \tau_i \) and denote \( n_{i,x} \) as the number of \( m_{i,x} \)'s accesses that are classified as an \( A \) or \( U \) at the LLC cache. Note that \( HB_k \) and \( CB_i \) include the memory blocks that meet the requirement in every program path that may be taken by the task.

In our system architecture, cache interference occurs only at the shared LLC cache when a cache line used by \( \tau_k \) is evicted by \( \tau_t \) and consequently causing reload overhead for \( \tau_k \). A cache line that may cause cache interference for \( \tau_k \) needs to satisfy at least two conditions:

(i) access to that cache line will result in a cache hit at the LLC cache in WCET analysis of \( \tau_k \),

(ii) the cache line may be used by \( \tau_t \).

From the above two conditions, we can analyze memory block accessing that may cause interference. The first condition implies that only accessing to \( HB_k \) may cause cache interference for \( \tau_k \), while the second condition indicates that accessing to \( CB_i \) by \( \tau_t \) may interfere with \( \tau_i \). Furthermore, cache interference occurs only if \( \tau_k \) accesses memory blocks in \( HB_k \) and \( \tau_i \) accesses memory blocks in \( CB_i \), concurrently, and those memory blocks have the same cache index.

We use \( I_k^{rec} \) to represent the upper bound on the shared cache interference imposed on \( \tau_k \) by only one job execution of \( \tau_t \).

Suppose the indexes of the LLC cache range from 0 to \( N - 1 \), we can derive \( N \) subsets of \( HB_k \) according to the mapping function \( \text{id} \) that maps a memory address to the cache line index at the LLC cache as follows:

\[
\tilde{m}_{k,u} = \{m_{k,x} \in HB_k|\text{id}(m_{k,x}) = u\}, (0 \leq u < N, u \in \mathbb{N}).
\]

We define the characteristic function of a set \( A \) which indicates membership of an element \( x \) in \( A \) as

\[
\chi_A(x) = \begin{cases} 
1 & x \in A \\
0 & \text{otherwise}
\end{cases}
\]

Let \( N_{k,u} \) represent the number of hit accesses to the \( u \)th cache line by \( \tau_k \) without cache interference. \( N_{k,u} \) equals to the total number of access to the \( HB_k \) mapping to the \( k \)th cache line

\[
N_{k,u} = \sum_{x=1}^{p} n_{k,x} \chi_{\tilde{m}_{k,u}}(m_{k,x}).
\]

Similarly, we divide \( CB_i \) into \( N \) subsets by

\[
\tilde{e}_{i,u} = \{m_{i,x} \in CB_i|\text{id}(m_{i,x}) = u\}, (0 \leq u < N, u \in \mathbb{N}).
\]

The number of accesses to the \( k \)th cache line by \( \tau_i \) is bounded by

\[
N_{i,u} = \sum_{x=1}^{q} n_{i,x} \chi_{\tilde{e}_{i,u}}(m_{i,x}).
\]

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on October 22,2020 at 16:17:27 UTC from IEEE Xplore. Restrictions apply.
Cache interference can only happen among memory blocks that are in the same subset that maps to the same cache line. For the $i$th cache line, $\tau_k$ can be interfered at most $N_{i,u}$ times and $\tau_i$ can interfere at most $N_{i,u}$ times. The following formula gives an upper bound on the number of cache misses by accessing the HBs for task $\tau_k$

$$S(\tau, \tau_k) = \sum_{i=0}^{N-1} \min(N_{i,u}, N_{k,u}).$$

Suppose the penalty for an LLC cache miss is a constant, $C_{\text{miss}}$, then $I_{i,k}^{\text{sc}}$ can be calculated by

$$I_{i,k}^{\text{sc}} = S(\tau_i, \tau_k)C_{\text{miss}}.$$

**Lemma 5.** The shared cache interference imposed on $\tau_k$ by only one job execution of $\tau_i$ can be bounded and $I_{i,k}^{\text{sc}} = S(\tau_i, \tau_k)C_{\text{miss}}$.

**Proof.** The lemma holds as discussed above. □

The computation of $I_{i,k}^{\text{sc}}$ only takes the memory accesses of $\tau_k$ and $\tau_i$ as input, so $I_{i,k}^{\text{sc}}$ only depends on memory accesses of $\tau_k$ and $\tau_i$. Given a taskset, $I_{i,k}^{\text{sc}}$ can be computed. In the following discussion, we assume $I_{i,k}^{\text{sc}}$ is known.

Lemma 5 gives an upper bound on cache interference for $\tau_i$ imposed by only one job of $\tau_i$. It is possible that more than one job of $\tau_i$ interfere with $\tau_k$. We denote the number of jobs of $\tau_i$ that interfere with $\tau_k$ as $N_{i,k}$.

**Lemma 6.** The total cache interference $t_{\text{sc}}$ exhibited from $N_{i,k}$ jobs of $\tau_i$ is bounded by $N_{i,k}I_{i,k}^{\text{sc}}$.

**Proof.** For $N_{i,k}$ jobs of $\tau_i$, the total number of accesses to each memory block $m_{i,z}$ is bounded by $N_{i,k}m_{i,z}$. Thus, the execution of $N_{i,k}$ jobs of $\tau_i$ accesses the $k$th cache line also at most $N_{i,k}N_{i,u}$ times. From the proof of Lemma 5, the upper bound of the total cache interference exhibited by $\tau_k$ from $N_{i,k}$ jobs of $\tau_i$ is $\sum_{i=0}^{N-1} \min(N_{i,k}N_{i,u}, N_{k,u})C_{\text{miss}}$

$$N_{i,k}I_{i,k}^{\text{sc}} = N_{i,k} \sum_{i=0}^{N-1} \min(N_{i,u}, N_{k,u})C_{\text{miss}}$$

$$= \sum_{i=0}^{N-1} \min(N_{i,k}N_{i,u}, N_{k,u})C_{\text{miss}}$$

$$\geq \sum_{i=0}^{N-1} \min(N_{i,k}N_{i,u}, N_{k,u})C_{\text{miss}}.$$

□

### 4.3.2 IP Formulation

We can compute an upper bound of the maximum cache interference a task may exhibit during an execution window by introducing an Integer Programming (IP) formulation, which can be transformed to an integer linear programming formulation.

It is necessary to check the schedulability of the task-set without considering cache interference. If the task-set does not pass the initial schedulability test, there is no need to calculate the cache interference. Only if all tasks (including $\tau_i$) pass the schedulability test (without considering cache interference), the IP is solved to compute the upper bound on cache interference. Therefore, the IP formulation is based on the assumption that $\tau_i$ is schedulable without cache interference.

If $N_{i,k}$ jobs of $\tau_i$ are executing concurrently with $\tau_k$, the cache interference that $\tau_k$ causes on $\tau_i$ is bounded by $N_{i,k}I_{i,k}^{\text{sc}}$ according to Lemma 6. As a task may exhibit cache interference from more than one task during a job execution, the total cache interference for one job execution of $\tau_k$ is bounded by the sum of the contributions of all other tasks $\tau_j (i \neq k)$ in the task set $\tau$. Thus, the objective function of the IP formulation is

$$\max \sum_{i \neq k} N_{i,k}I_{i,k}^{\text{sc}}.$$ (13)

The IP formulation will have an unbounded solution without further constraints to the variable $N_{i,k}$. To get a bounded solution, we analyze the constraints on $N_{i,k}$. First, we define the concept of the execution window of a job.

**Definition 4.** The Execution Window (EW) of the $j$th job of $\tau_k$ ($J_k^j$) is time interval $[s_k^j, f_k^j]$ from the staring time to the finishing time of $J_k^j$.

Note that the length of an execution window may be larger than $C_k$, since the EW includes the cache interference. We use $C_k$ as the length of the EW because of the iterative computation which will be described later on.

$N_{i,k}$ reaches its minimal value when a job of $\tau_i$ starts to execute as soon as it is released and the execution finishes just before the start of the EW, as shown the case (a) in Fig. 10. Denoting $C_{\text{min}}$ as the smallest execution time of $\tau_i$, often called Best-Case Execution Time (BCET), we have the following constraint:

$$\forall i \neq k, \left[\max\left(0, C_k^i - T_i + C_{\text{min}}\right)\right] + \xi_i \leq N_{i,k},$$ (14)

where $\xi_i = \begin{cases} 1 & (C_k^i + C_{\text{min}} \mod T_i) - D_i + C_{\text{min}} > 0 \\ 0 & \text{otherwise} \end{cases}$.

Note that the last job of $\tau_i$ is released within the EW will interfere with $\tau_k$, since the last released job should start its execution $C_{\text{min}}$ before its relative deadline if the task is schedulable.

The maximum value of $N_{i,k}$ is taken when the first interfering job of $\tau_i$ finishes just after the start of the EW and the last interfering job of $\tau_i$ starts to execute at the time when it is released. Such a situation is depicted as case (b) in Fig. 10. Thus, we have the second constraint on $N_{i,k}$

$$\forall i \neq k, N_{i,k} \leq 1 + \left[\max\left(0, C_k^i - T_i + D_i\right)\right].$$ (15)

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

**Lemma 7.** If $\tau_i$ is schedulable without considering cache interference, $C_i$ computation capacity of the processing core is reserved for the execution of $J_i'$ during $[r_i', d_i']$. If $J_i'$ executes for $C_i' < C_i$, the processing core will be accumulatively idle (executing nothing, simply wasting the processing capacity for $\tau_i$) for at least $C_i - C_i'$ during $[r_i', d_i']$.

**Proof.** If $\tau_i$ satisfies the schedulability condition without considering cache interference: $\frac{W_i(0, C_i)}{m} + C_i < D_i$, the core on which $J_i'$ is executed spends most of $D_i - C_i$ in total for the execution of other interfering tasks during $[r_i', d_i']$. $J_i'$ is guaranteed to have $C_i$ computation capacity during $[r_i', d_i']$. 

The remaining computation capacity of a multicore processor with $m$ cores is $(m - 1)C_k'$ since one core is dedicated to the execution of $\tau_k$. Due to the limited computation capacity of the processor, the total execution of the tasks that may interfere with $\tau_k$ within the EW cannot exceed $(m - 1)C_k'$. Hence, we have the third constraint

$$\sum_{i \neq k} \max(0, N_{i,k} - 2)C_i \leq (m - 1)C_k'.$$

(16)

The objective function (13) together with three constraints on $N_{i,k}$, i.e., inequalities (14), (15) and (16), form our IP problem. Since $C_{i}^{\min}$ is a relatively small number, we take the extreme case: $C_{i}^{\min} = 0$. As task parameters such as $C_i, D_i, T_i$ are known, the optimal solution of the IP only depends on the length of $EW$. Thus, we use $I^{sc}(C_k')$ to denote the optimal value of the IP problem if $C_k'$ is used as the length of the $EW$ in the IP.

Note that inequalities (14) and (16) are based on the assumption that $\tau_i$ is schedulable. Thus, before solving the IP, we have to check the schedulability of the taskset assuming no cache interference between tasks, i.e., $I^{sc} = 0$.

**Computation Complexity of the IP.** The original IP can be easily transformed to an Integer Linear Programming (ILP) problem by introducing a new integer variable $y_{i,j}$ for each $N_{i,j}$ with two additional constraints: $y_{i,j} \geq 0$ and $y_{i,j} \geq N_{i,k} - 2$. Inequality (16) can be replaced by $\sum_{i \neq k} y_{i,k} C_i \leq (m - 1)C_k'$. In the transformed ILP problem, we have totally $2(n - 1)$ variables and $4(n - 1) + 1$ constraints. The complexity of the IP is the same as the complexity of solving the transformed ILP problem, which is $O(n6^{4n}/4n)$ [31]. Despite the exponential complexity, current LP solver implementations are very efficient and capable of solving realistic LP problem formulations. We will demonstrate this in Section 6.

## 5 Iterative Computation

Due to the presence of cache interference, a job may execute longer than $C_k$ on a multicore platform with shared caches. However, a larger execution time may introduce more cache interference, as illustrated in Fig. 11.

In Fig. 11a, if the job of $\tau_k$ executes for $C_k'$, only one job of $\tau_i$ interferes with $\tau_k$. In Fig. 11b, if the job of $\tau_k$ executes for a larger execution time, say $C_k' + I^{sc}(C_k')$, two jobs of $\tau_i$ could possibly interfere with $\tau_k$, which potentially may increase the cache interference exhibited by $\tau_k$. This example suggests an iterative method is needed to find an upper bound on the cache interference.

**Lemma 8.** $I^{sc}(C_k')$ is non-decreasing with respect to $C_k$.

Lemma 8 is explained by the above example. We give a sufficient condition for a certain value that can be used as an upper bound on cache interference.

**Lemma 9.** if $\exists C_k' \geq C_k$ such that $C_k' = C_k + I^{sc}(C_k')$, then $I^{sc}(C_k') = I^{sc}(C_k)$. According to Lemma 8, given an execution window of $t_k$ that is no more than $C_k + I^{sc}(C_k')$, the cache interference exhibited by $\tau_k$ is not larger than $I^{sc}(C_k')$. Therefore, $I^{sc}(C_k')$ is the upper bound on cache interference for $\tau_k$. By definition, $I^{sc}(C_k') = I^{sc}(C_k')$.

We now derive the iterative algorithm, called $Cache\text{Interference}(t, m)$ to compute an upper bound on cache interference for each task $t_k \in \tau$:

- Since the constraints of our IP formulation assume the taskset is schedulable, we first assess the schedulability of the taskset assuming no cache interference between each task. Only if all tasks pass schedulability test, the following steps will be taken.
- $C_k'$ is initialized with $C_k$ and an upper bound value on the cache interference $I^{sc}(C_k')$ is created which is initially set to zero.
- By solving the IP, we compute a new upper bound of the cache interference $I^{sc}(C_k')$. If the new upper bound of cache interference is the same as the old upper bound, the $I^{sc}(C_k')$ is the final upper bound of $t_k$. Otherwise, another round of computing the upper bound on cache interference is performed using the upper bound derived at the previous iteration. The iteration for $t_k$ stops either if no update on $I^{sc}(C_k')$ is possible anymore or if the computed $I^{sc}(C_k')$ is large enough to make $t_k$ unschedulable.
- The previous steps are repeated for every task in $\tau$.

A more formal version of the $Cache\text{Interference}(t, m)$ algorithm is given by Pseudocode 1. The algorithm returns $I^*$ which includes the upper bounds on cache interference $I^{sc}(C_k')$ for each task $t_k$ and $C^*$ which includes the upper bounds on the execution length $C_k$ for each $t_k$. If $I^*$ and $C^*$ are empty, the taskset is not schedulable.

Since the solution of the IP is non-decreasing with respect to $C_k'$ according to Lemma 8 and one termination condition is $C_k' \geq D_k$, the termination of the iterative algorithm is guaranteed.

Before presenting the final theorem to check the schedulability of the task set, we define the following notations.

---

*Fig. 11. More cache interference if $\tau_k$ executes for a longer time.*
Pseudocode 1. CacheInterference($\tau$, $m$)

1: Input: Task parameters, number of cores: $m$
2: $I^* \leftarrow$ empty list, used to store $I^w(C^*_t)$ for each task
3: $C^* \leftarrow$ empty list, used to store $C^*_t$ for each task
4: for all $t_k \in \tau$ do
5: update $\leftarrow$ true, $I^o_k \leftarrow 0, I^n_k \leftarrow 0$
6: $C^*_k \leftarrow C_k$
7: while update do
8: $I^p_k \leftarrow I^p_k$
9: $I^n_k \leftarrow$ Solution of IP with $C^*_i$ as the EW
10: if $I^n_k = I^p_k$ or $C^*_k \geq D_k$ then
11: update $\leftarrow$ false
12: end if
13: end while
14: Add $I^n_k$ to $I^*$
15: Add $C^*_k$ to $C^*$
16: end for
17: return $I^*, C^*$

We denote $U(t_k)$ as task $t_k$’s utilization taking shared cache interference into account, $U(t)$ is defined by

$$U_i = \frac{C^*_i}{T_i}.$$  

The utilization of task set $\tau$, denoted by $U(\tau)$, is defined by

$$U(\tau) = \sum_{t_k \in \tau} U_i = \sum_{t_k \in \tau} \frac{C^*_i}{T_i}.$$  

We sort all $C^*_i$ in a non-increasing order, and use $\Delta^{m-1}$ to denote the sum of the first ($m-1$) elements in this list, so

$$\Delta^{m-1} = \sum_{\text{the (m-1) largest}} C^*_i.$$  

For task $t_k$, we also define a constant $L_k$ by

$$L_k = \frac{\sum_{t_k \in \tau} C^*_i + \Delta^{m-1}}{m - U(\tau)} - S_k.$$  

(17)

We propose the following Theorem to check the schedulability of the task set.

**Theorem 2.** A task set $\tau$ is schedulable with the EDF or FP scheduling policy on a multicore platform composed of $m$ identical cores with shared caches if for each task $t_k \in \tau$ and $0 \leq A_k \leq L_k$,

1. $\exists C^*_k \geq C_k$ such that $C^*_k = C_k + I^w(C^*_t)$,
2. $\frac{D_k}{m} + C^*_k < D_k + A_k$.

**Proof.** From (1), $I^w_k$ is bounded and $I^w_k = I^w(C^*_k)$ according to Lemma 9.

From Lemma 4, $I^p_k = \Omega_k(C^*_k)$.

$$\forall A_k \geq 0, \text{ if } \frac{D_k}{m} + C^*_k < D_k + A_k \text{ then } I^p_k + C^*_k < A_k + D_k.$$  

Theorem 2 follows from Theorem 1.

We further prove that if condition (2) is to be violated for any $A_k$, then $\exists A_k, \frac{D_k}{m} + C^*_k \geq D_k + A_k$.

$$\Rightarrow \Omega_k \geq m(D_k + A_k - C^*_k)$$

$$\Rightarrow (A_k + S_k)U(\tau) + \sum_{t_k \in \tau} C^*_i + \Delta^{m-1} \geq m(S_k + A_k).$$

If condition (2) is to be violated for any $A_k$, then $\exists A_k, \frac{D_k}{m} + C^*_k \geq D_k + A_k$.

$$\Rightarrow \Omega_k \geq m(D_k + A_k - C^*_k)$$

$$\Rightarrow (A_k + S_k)U(\tau) + \sum_{t_k \in \tau} C^*_i + \Delta^{m-1} \geq m(S_k + A_k).$$

Solve the above inequality for $A_k$, we have

$$A_k \leq \frac{\sum_{t_k \in \tau} C^*_i + \Delta^{m-1}}{m - U(\tau)} - S_k = L_k.$$  

This tells us the range of $A_k$ that should be tested. $\square$

Finally, we give the procedure CheckSchedulability($\tau$, $m$) to perform the schedulability test, as illustrated by Pseudocode 2.

Pseudocode 2. CheckSchedulability($\tau$, $m$)

1: Input: Task parameters, number of cores: $m$
2: $I^*, C^* \leftarrow$ CacheInterference($\tau$, $m$)
3: for all $t_k \in \tau$ do
4: calculate $L_k$ by Equation (17)
5: for all $A_k \in [0, L_k]$ do
6: $\Omega_k \leftarrow$ calculation of Equation (12) using $C^*, A_k$
7: if $\frac{D_k}{m} + C^*_k \geq D_k + A_k$ then
8: return Unschedulable
9: end if
10: end for
11: end for
12: return Schedulable

**Computational Complexity.** Let $n$ denote the number of tasks in the task set. For $t_k$, let $I^w_{\text{min}}$ be the smallest difference between cache interference caused by one job of $t_i$ and $t_j$, i.e., $I^w_{\text{min}} = \min(I^w_k - I^w_t)$. The iterative algorithm takes at most $\eta = \max_k \frac{(D_k - C_k)}{(I^w_{\text{min}} - I^w_k)}$ iterations to terminate since $C^*_k$ either stays the same or increases at least a distance $I^w_{\text{min}}$ in each iteration. Thus, the complexity of the iterative algorithm to compute the upper bound on cache interference is $O(n^2 m^2 \ln 4n)$. The complexity of computing $L_k, \Omega_k$ is polynomial. Therefore, the complexity to perform the schedulability test is $O(n^2 m^2 \ln 4n)$.
6 EXPERIMENTS

In this section, we systematically generate synthetic workloads to evaluate the performance of the proposed schedulability test for $E_D^{np}$ and $F^{np}$ in terms of acceptance ratio. More specifically, we will quantify the effects of cache interference on the schedulability of the generated tasksets. We will also compare the schedulability performance of $E_D^{np}$ against $F^{np}$ over randomly generated tasksets.

The experiments have been performed varying i) the probability of two tasks having cache interference on each other: $P$ ($P = 0.1, 0.2, 0.3$ or $0.4$), ii) the cache interference factor $IF$ ($IF = 0, 0.3, 0.6$ or $0.9$), iii) the number of cores $m$ ($m = 2, 4$ or $8$), iv) total task utilization $U_{tot}$ ($U_{tot}$ from $0.1$ to $m - 0.1$ with steps of $0.2$). Given those three parameters, we have generated $20000$ tasksets in each experiment. The number of tasks $n$ in each taskset is $10$, i.e., $n = 10$. 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 Randfixed-sum [32] 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 $T_k$, $T_k$ is uniformly distributed over the interval $[100, 200]$. The WCET of $T_k$ 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 task having cache interference is $P$. If two tasks $T_k$ and $T_i$ interfere with each other, $I_{ik}^{np}$ is generated as $I_{ik}^{np} = IF \times \min(0.5C_i, 0.5C_k)$.

In each experiment, we measure the number of schedulable tasksets that pass the proposed schedulability test. The acceptance ratios, which is the number of schedulable tasksets divided by the total number of tasksets (20000), are shown in Figs. 12 and 13 for $E_D^{np}$ and $F^{np}$, respectively.

Fixing $m = 4$, $n = 10$, $IF = 0.3$, Figs. 12a and 13a illustrate the acceptance ratio with different $P$ for $E_D^{np}$ and $F^{np}$, respectively. With the same $U_{tot}$, the acceptance ratio for both $E_D^{np}$ and $F^{np}$ decreases as $P$ increases because a larger $P$ indicates more tasks in the taskset could interfere with each other, which may potentially increase the upper bound on cache interference for each task. Fixing $P$, it can be observed that the acceptance ratio of $E_D^{np}$ is higher than $F^{np}$ when $U_{tot} \in [1, 2.5]$. For example, when $P = 0.2$, $IF = 0.3$ and $U_{tot} = 1.7$, $60.1\%$ of tasksets are schedulable by $E_D^{np}$ while $F^{np}$ schedules $50.45\%$ of the generated tasksets.

Figs. 12b and 13b show the acceptance ratio achieved by $E_D^{np}$ and $F^{np}$, respectively, for the cases $IF = 0, 0.3, 0.6, 0.9$, fixing $m = 4$, $n = 10$, $P = 0.4$. The red line with $IF = 0$ represents the acceptance ratio when tasks have no cache interference. Evidently, the acceptance ratios with a lower $IF$ are better than those with a larger $IF$. As we increase $IF$ with the same amount, the average acceptance ratio decreases in a slower fashion. However, it does not indicate that a lower bound on the average acceptance ratio is possible since the cache interference gets larger as $IF$ increases, eventually making the interfered tasks unschedulable. Fixing $IF$, it is also clear that the acceptance ratio achieved by $E_D^{np}$ is better than $F^{np}$ when $U_{tot} \in [0.7, 2.5]$. For example, when $P = 0.4$, $IF = 0.6$ and $U_{tot} = 1.1$, $66.9\%$ of tasksets are schedulable by $E_D^{np}$ while $F^{np}$ schedules $59.3\%$ of the generated tasksets.

Figs. 12c and 13c illustrate the acceptance ratio with respect to the number of cores for $E_D^{np}$ and $F^{np}$, respectively. In
the two figures, the acceptance ratio for tasks having no cache interference are also plotted. Instead of using $U_{tot}$ as horizontal axis, we scale the horizontal axis with $\frac{U_{tot}}{m}$ for $m = 2, 4$. It is worth noting that an execution platform with fewer cores is more efficient in terms of acceptance ratio than those with more cores. This is due to the fact that the pessimism of the analysis of processor-contention interference and shared cache interference becomes worse when the number of cores increases. However, for processors with different numbers of cores scheduled by $EDF_{np}$ (or $FP_{np}$), the difference in the acceptance ratio of scheduling between the baseline (tasks having no cache interference, $P = 0$) and tasks having cache interference is almost similar.

**Average Execution Time.** We measured the execution time of running the proposed schedulability test with different task-set scales. The executions are conducted on an Intel Xeon processor using only one core running at 2.4 GHz. On average, it takes 0.13 seconds to check the schedulability of tasksets consisting of 10 tasks, 0.27 seconds for tasksets with 20 tasks, and 0.56 seconds for tasksets with 30 tasks.

7 Conclusion
In this paper, we developed a new schedulability analysis of global scheduling ($EDF_{np}$ and $FP_{np}$) for real-time multicore systems with shared caches. We constructed an integer programming formulation that can be transformed to an integer programming formulation to calculate the upper bound on cache interference exhibited by a task during a given execution window. Using this integer formulation, we subsequently proposed an iterative algorithm to obtain an upper bound on the shared cache interference a task may exhibit during one job execution. We derived a new schedulability condition by integrating the upper bound on the cache interference into the schedulability analysis. A set of experiments has been performed using our proposed schedulability analysis to demonstrate the effects of cache interference for a range of different tasksets. We also compared the schedulability performance of $EDF_{np}$ against $FP_{np}$ in the presence of cache interference. Our empirical evaluations showed that $EDF_{np}$ is better than $FP_{np}$ in terms of tasksets deemed schedulable. As for future work, we plan to extend our schedulability analysis to real-time multicore systems with shared caches that use preemptive task scheduling.

Acknowledgments
The research of this article was supported by Netherlands Organisation for Scientific Research under Project No. 12696 and the University of Amsterdam.

References


Jun Xiao (Member, IEEE) received the BE degree in automation and control engineering from Nanchang University, Nanchang, China, in 2012, the MS degree from the University of Trento and Scuola Superiore Sant’Anna, Pisa, Italy, in 2014, and the PhD degree in computer science from the University of Amsterdam, Amsterdam, The Netherlands, in October, 2019. He is a postdoc researcher with the University of Amsterdam. His research interests include the fields of embedded and real-time systems, schedulability analysis, and computer architecture.

Sebastian Altmeyer (Member, IEEE) received the PhD degree in computer science from Saarland University, Saarbrücken, Germany, in 2012 with a thesis on the analysis of preemptively scheduled hard real-time systems. He is currently an assistant professor (Universitair Docent) with the University of Amsterdam. From 2013 to 2015, he has been a postdoctoral researcher with the University of Amsterdam, and from 2015 to 2016 with the University of Luxembourg. In 2015, he has received an NWO Veni Grant on the timing verification of real-time multicore systems. He has been a program chair of ECRTS 2018 and has served on many conferences on real-time embedded systems, including RTSS, RTAS, RTNS, DATE, and DAC. His research focuses various aspects of the design, analysis and verification of hard real-time systems, with a particular interest in timing verification and multicore architectures.

Andy D. Pimentel (Senior Member, IEEE) received the MSc and PhD degrees in computer science from the University of Amsterdam, Amsterdam, The Netherlands. He is currently an associate professor with the System and Network Engineering Lab, University of Amsterdam. His research interests include system-level modeling, simulation, and exploration of (embedded) multicore and manycore computer systems with the purpose of efficiently and effectively designing and programming these systems. He is a co-founder of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). He has (co)authored more than 100 scientific publications and is an associate editor of the Elseviers Simulation Modeling Practice and Theory as well as the Springer’s Journal of Signal Processing Systems. He served as the general chair of HiPEAC’15, as Local Organization co-chair of ESWEEK’15, and as program (vice-)chair of CODES+ISSS in 2016 and 2017. Furthermore, he has served on the TPC of many leading (embedded) computer systems design conferences, such as DAC, DATE, CODES+ISSS, ICCD, ICCAD, FPL, SAMOS, and ESTIMedia.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.