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

• *The authors are with Informatics Institute, University of Amsterdam, 1098XH Amsterdam, The Netherlands. E-mail: {J.Xiao, S.J.Altmeyer, A.D.Pimentel}@uva.nl.*

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 multicores with shared caches. 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 $\times$ 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 τ of n periodic or sporadic real-time tasks $\{\tau_1, \tau_2, \ldots \tau_n\}$ to be scheduled on a multicore processor. Each task $\tau_k = (C_k, D_k, T_k) \in \tau$ is characterized by a worst-case

computation time C_k, a period or minimum inter-arrival time T_k, and a relative deadline D_k. All tasks are considered to be deadline constrained, i.e., the task relative deadline is less or equal to the task period: $D_k \leq T_k$.

We further assume that all those tasks are independent, i.e., they have no shared variables, no precedence constraints, and so on. Moreover, jobs of any task cannot be executed at the same time on more than one core. A task τ_k is a sequence of jobs J_k^j, where j is the job index. We denote the arrival time, starting time, finishing time and absolute deadline of a job j as r_k^j, s_k^j, f_k^j and d_k^j, respectively. Note that the goal of a real-time scheduling algorithm is to guarantee that each job will complete before its absolute deadline: $f_k^j \leq d_k^j = r_k^j + D_k$.

As explained, it is difficult to accurately estimate C_k considering cache interference of other tasks executing concurrently. It should be pointed out that C_k in this paper refers to the WCET of task k, assuming task k is the only task executing on the 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_{np}) and Non-Preemptive Fixed Priority (FP_{np}).

EDF_{np} assigns a priority to a job according to the absolute deadline of that job. A job with an earlier absolute deadline has higher priority than others with a later absolute deadline. Since each job's absolute deadline changes over time, the priority of a task changes dynamically.

For FP_{np} scheduling, a fixed priority P_k is assigned to each task τ_k ($k = 1, 2, \ldots n$). As each task has a unique priority, we use $hp(k)$ to denote the set of tasks with higher priorities than τ_k, and $hep(k) = hp(k) \cup \{\tau_k\}$ the set of tasks whose priorities are not lower than τ_k. Similarly, $lp(k)$ is the set of tasks with lower priorities than τ_k and $lep(k) = lp(k) \cup \{\tau_k\}$ the set of tasks whose priorities are not higher than τ_k.

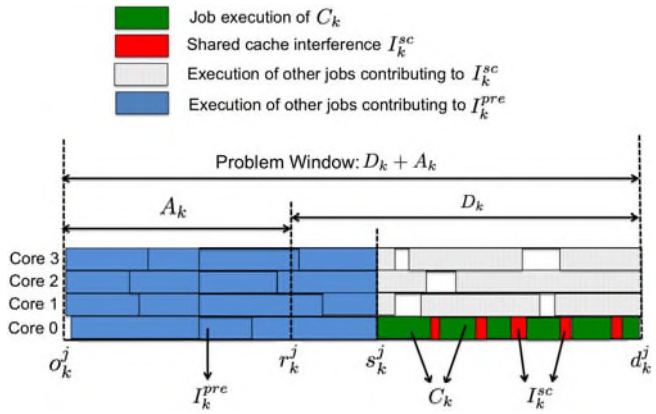

Fig. 1. Overview of the schedulability analysis that accounts for cache interference.

The EDF_{np} and FP_{np} 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^j of a task τ_k. Let o_k^j denote the latest time-instant no later than r_k^j ($o_k^j \leq r_k^j$) at which at least one processor is idle and let $A_k = r_k^j - o_k^j$. As all processors are idle when the system starts, there always exists such a o_k^j. The time interval $[o_k^j, d_k^j]$ can be divided into two parts $[o_k^j, s_k^j]$ and $[s_k^j, d_k^j]$.

As shown in Fig. 1, a job J_k^j of task τ_k exhibits two kinds of interferences during $[o_k^j, d_k^j]$. The first interference is called processor-contention interference, denoted by I_k^{pre}. It is the cumulative length of all intervals over $[o_k^j, s_k^j]$ in which all the processing cores are busy executing jobs other than J_k^j. We define the interference $I_{i,k}^{pre}$ of a task τ_i on a task τ_k over the interval $[o_k^j, s_k^j)$ as the cumulative length of all intervals in which τ_i 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_k^{sc}. We also define the interference $I_{i,k}^{sc}$ as the cumulative length of all extra execution delays of τ_k caused by shared cache accesses between task τ_i and task τ_k.

Furthermore, we define the upper bound on processor-contention interference as $\bar{I}_k^{pre}$ and similarly the upper bound on shared cache interference as $\bar{I}_k^{sc}$.

Note that the processor-contention interference I_k^{pre} occurs during $[o_k^j, s_k^j]$, so I_k^{pre} depends on A_k and the length of $[r_k^j, s_k^j]$. While the shared cached interference I_k^{sc} occurs only during τ_k's execution. We will present the derivation of $\bar{I}_k^{sc}$ in the next section and it can be shown that $\bar{I}_k^{sc}$ does not depend

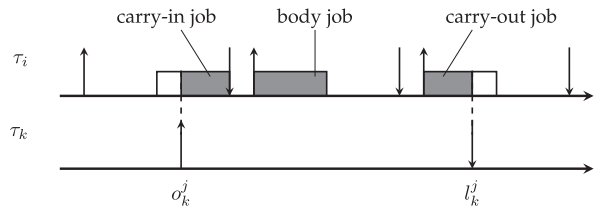

Fig. 2. Three types of contribution jobs and problem window.

on A_k and the length of $[r_k^j, s_k^j]$. Let us now assume $\bar{I}_k^{sc}$ is known.

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

As the processor-contention interference only occurs before the start of the τ_k's execution, we restrict I_k^{pre}, $I_{i,k}^{pre}$ and $\bar{I}_k^{pre}$ to the time interval $[o_k^j, l_k^j]$.

By construction, we have the first schedulability test for τ.

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

$$\bar{I}_k^{pre} + C_k + \bar{I}_k^{sc} < D_k + A_k.$$

4.2 Computation of $\bar{I}_k^{pre}$

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

Lemma 1. *The processor-contention interference that a task τ_i causes on a task τ_k in $[o_k^j, l_k^j)$ is never greater than the workload of τ_i in $[o_k^j, l_k^j)$*

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

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

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

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

carry-in job: a job with its release time earlier than o_k^j but with its deadline earlier than l_k^j;

body job: a job with both its release time and its deadline in $[o_k^j, l_k^j]$;

carry-out job: a job with its release time in $[o_k^j, l_k^j]$, but with its deadline later than l_k^j.

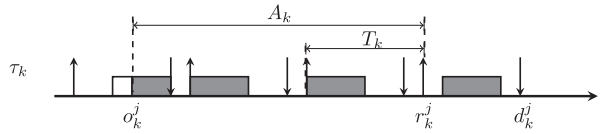

Fig. 3. The densest possible packing of jobs of τ_i without carry-in job, if $i = k$.

As shown in Fig. 2, the worst-case workload of τ_i occurs when a carry-in job (if τ_i 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 $W_{i,k}^n$ to denote an upper bound of τ_i's workload in $[o_k^j, l_k^j]$ if τ_i has no carry-in job, and use $W_{i,k}^c$ to denote an upper bound of τ_i's workload if τ_i has a carry-in job.

Following the approach in [19], we derive a tighter upper bound on W_i^n and W_i^c 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 $W_{i,k}^n$ 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_i^j to cause interference to J_k^j is $r_i^j < r_k^j$, i.e.,J_i^j must be released earlier than J_k^j; if $D_i \leq D_k$, the necessary condition for J_i^j to cause interference to J_k^j is $d_i \leq d_k$, i.e., J_i's absolute deadline must be no later than that of J_k.*

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

Since τ_i has no carry-in jobs in this case, the worst case of $W_{i,k}^n$ occurs when the first job of τ_i is released at time o_k^j. The next jobs of τ_i are then released periodically every T_i time units. Thus, $W_{i,k}^n$ 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 $[o_k^j, r_k^j]$ contribute to processor-contention interference and the number of τ_i's body instances is $\left\lfloor \frac{A_k}{T_k} \right\rfloor$. So we have

$$W_{i,k}^{n1} = \left\lfloor \frac{A_k}{T_k} \right\rfloor C_k^*. \tag{1}$$

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

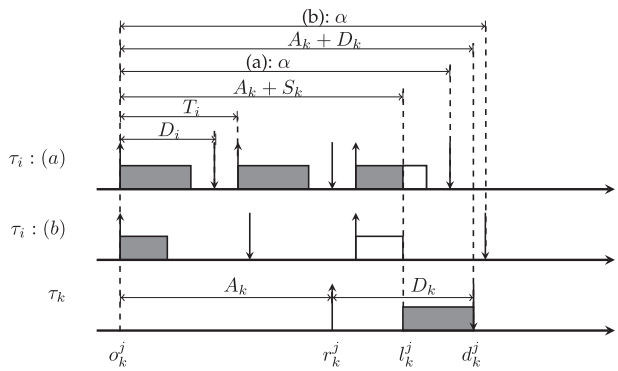

Fig. 4. The densest possible packing of jobs of τ_i without carry-in job and $D_i \leq D_k$. Case (a): $\alpha \leq A_k + D_k$, Case (b): $\alpha > A_k + D_k$.

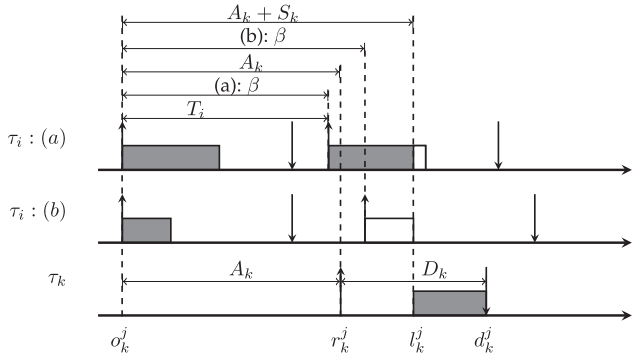

Fig. 5. The densest possible packing of jobs of τ_i without carry-in job and $D_i > D_k$. Case (a): $\beta < A_k$, Case (b): $\beta \geq A_k$.

We use α to denote the distance between o_k^j and the deadline of τ_i's carry-out job, $\alpha = \left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor T_i + D_i$. The deadline of τ_i's carry-out job is $o_k^j + \alpha$.

(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 $\min(C_i^*, (A_k + S_k) \bmod T_i)$. In this case, we have

$$W_{i,k}^{n_2} = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor C_i^* + \min(C_i^*, (A_k + S_k) \bmod T_i). \quad (2)$$

(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

$$W_{i,k}^{n_3} = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor C_i^*. \quad (3)$$

3) $D_i > D_k$. Fig. 5 shows the worst case of $W_{i,k}^n$ for $D_i > D_k$. The number of body jobs of τ_i is $\left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor$. By Lemma 2, a job of τ_i can interfere with J_k^j only if its release time is earlier than r_k^j. We use β to denote the distance between o_k^j and the release time of τ_i's carry-out job, $\beta = \left\lfloor \frac{A_k+S_k}{T_i} \right\rfloor T_i$.

(3.A) If $A_k = 0$, then $o_k^j = r_k^j$. Since $D_i > D_k$, any task instance released no earlier than o_k^j has a deadline later than d_k^j, so, $W_{i,k}^n = 0$.

(3.B) If $\beta < A_k$, shown as case (a) in Fig. 5. The contribution of τ_i's carry-out job is bounded by $\min(C_i^*, (A_k + S_k) \bmod T_i)$. $W_{i,k}^n$ is computed by Equation (2).

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

By the discussions above, we can compute $W_{i,k}^n$ for EDF_{np} by

$$W_{i,k}^n = \begin{cases} 0 & D_i > D_k \wedge A_k = 0 \\ W_{i,k}^{n_1} & i = k \\ W_{i,k}^{n_2} & (i \neq k \wedge D_i \leq D_k \wedge \alpha \leq A_k + D_k), \\ & \vee (D_i > D_k \wedge \beta < A_k) \\ W_{i,k}^{n_3} & otherwise \end{cases} \quad (4)$$

where $W_{i,k}^{n_1}$, $W_{i,k}^{n_2}$, $W_{i,k}^{n_3}$ are defined in Equations (1), (2) and (3) respectively.

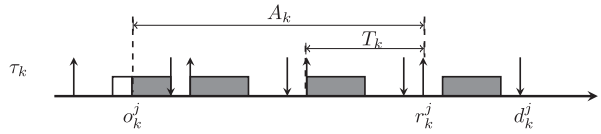

Fig. 6. The densest possible packing of jobs of τ_i with carry-in job, if $i = k$.

4.2.2 Upper Bound on $W_{i,k}^c$ for EDF_{np}

We now compute the upper bound on $W_{i,k}^c$ by four cases: (1) $i = k$, (2) $D_i \leq D_k$ and $S_i > C_k^*$ (3) $D_i > D_k$ and $S_i \geq C_k^*$ (4) the remaining cases.

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

$$W_{i,k}^{c_1} = \left\lfloor \frac{A_k}{T_k} \right\rfloor C_k^* + \min(C_k^*, \max(0, (A_k \bmod T_k) - T_k + D_k)). \quad (5)$$

2) $D_i \leq D_k \wedge S_i > C_k^*$. Shown as case (a) in Fig. 7, the worst case of $W_{i,k}^c$ occurs when τ_i's last released instance has its deadline at d_k^j. The number of τ_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 $\min(C_i^*, (A_k + D_k) \bmod T_i)$. So, we have

$$W_{i,k}^{c_2} = \left\lfloor \frac{A_k + D_k}{T_i} \right\rfloor C_i^* + \min(C_i^*, (A_k + D_k) \bmod T_i). \quad (6)$$

3) $D_i > D_k \wedge S_i \geq C_k^*$. Case (b) in Fig. 7 shows the worst case of $W_{i,k}^c$. By Lemma 2, τ_i's job can interfere with J_k^j only if its release time is earlier than r_k^j. So, the worst case of $W_{i,k}^c$ occurs when one of τ_i's instances is released at $r_k^j - 1$.

(3.A) If $A_k > 0$, the number of τ_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 $\mu = \min(C_i^*, \max(0, (A_k - 1) \bmod T_i - (T_i - D_i)))$.

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

$$W_{i,k}^{c_3} = \begin{cases} C_i^* - 1 & A_k = 0 \\ \left(\left\lfloor \frac{A_k-1}{T_i} \right\rfloor + 1\right) C_i^* + \mu & A_k > 0 \end{cases}. \quad (7)$$

4) For the remaining cases, i.e., $(D_i \leq D_k \wedge S_i \leq C_k^*) \vee (D_i > D_k \wedge S_i < C_k^*)$, the worst case of $W_{i,k}^c$ occurs

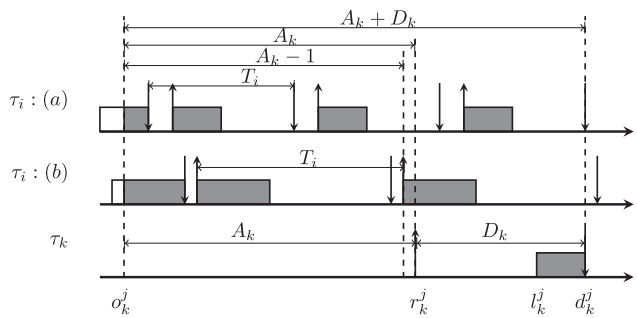

Fig. 7. The densest possible packing of jobs of τ_i with carry-in job. Case (a): $D_i \leq D_k \wedge S_i > C_k^*$, Case (b):$D_i > D_k \wedge S_i \geq C_k^*$.

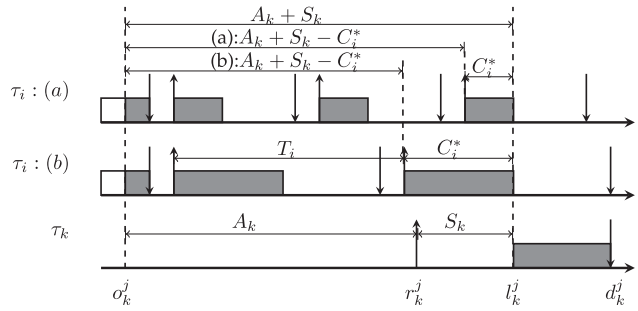

Fig. 8. The densest possible packing of jobs of τ_i with carry-in job. Case (a): $D_i \le D_k \wedge S_i \le C_k^*$, case (b):$D_i > D_k \wedge S_i < C_k^*$.

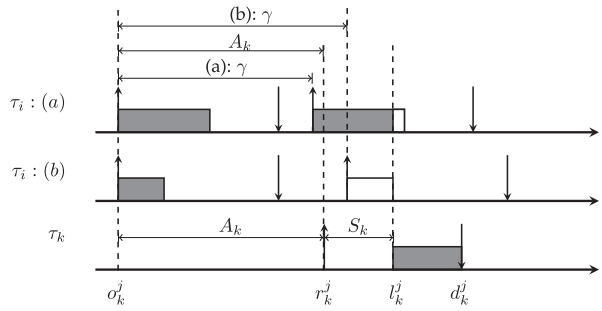

Fig. 9. The densest possible packing of jobs of τ_i with carry-in job. Case (a): $\gamma < A_k$, Case (b):$\gamma \ge A_k > 0$.

when one of τ_i's instances is released at $l_k^j - C_i^*$, as shown in Fig. 8.

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

(4.B) If $A_k + S_k > C_i^*$, the number of τ_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 $\nu = \min(C_i^*, \max(0, (A_k + S_k - C_i^*) \bmod T_i - (T_i - D_i)))$

$$W_{i,k}^{c_4} = \begin{cases} A_k + S_k & A_k + S_k \le C_i^* \\ \left(\left\lfloor \frac{A_k + S_k - C_i^*}{T_i} \right\rfloor + 1 \right) C_i^* + \nu & A_k + S_k > C_i^* \end{cases}. \tag{8}$$

By the discussion above, we can compute $W_{i,k}^c$ for EDF_{np} by

$$W_{i,k}^c = \begin{cases} W_{i,k}^{c_1} & i = k \\ W_{i,k}^{c_2} & i \ne k \wedge D_i \le D_k \wedge S_i > C_k^* \\ W_{i,k}^{c_3} & D_i > D_k \wedge S_k \ge C_i^* \\ W_{i,k}^{c_4} & otherwise \end{cases}, \tag{9}$$

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 $W_{i,k}^n$ for FP_{np}

The following lemma describes the condition of processor-contention interference on τ_k caused by lower-priority tasks in $lp(k)$ for FP_{np}.

Lemma 3. *For FP_{np}, a task instance J_i^j of $\tau_i \in lp(k)$ can interfere with J_k^j only if J_i^j is released before r_k^j.*

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

1) $i = k$. The worst-case workload is the same as in the case of EDF_{np}, thus $W_{i,j}^n$ can be computed by Equation (1).
2) $\tau_i \in hp(k)$. The worst-case workload of task τ_i occurs when a job of τ_i arrives at o_k^j, as shown in case (a) in Fig. 4. $W_{i,j}^n$ can be computed using Equation (2).
3) $\tau_i \in lp(k)$. The worse case of $W_{i,k}^n$ occurs when one of τ_i's instances is released at o_k^j. The number of body jobs of τ_i is $\left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor$. Let γ be the distance between o_k^j and the release time of τ_i's last instance. So $\gamma = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor$.
   (3.A) If $A_k = 0$, then $o_k^j = r_k^j$, according to Lemma 3, $W_{i,k}^n = 0$.
   (3.B) If $\gamma < A_k$, τ_i's last job is released earlier than r_k^j, as shown in Fig. 9 case (a), its contribution is bounded by $\min(A_k + S_k \bmod T_i, C_i^*)$. In this case, $W_{i,k}^n$ is computed by Equation (2).
   (3.C) If $\gamma \ge A_k > 0$, as shown in case (b) of Fig. 9, the contribution of the last released job of τ_i is 0. In this case, $W_{i,k}^n$ can be computed by Equation (3).

By the above discussion, we can compute $W_{i,k}^n$ by

$$W_{i,k}^n = \begin{cases} 0 & \tau_i \in lp(k) \wedge A_k = 0 \\ W_{i,k}^{n_1} & i = k \\ W_{i,k}^{n_2} & \tau_i \in hp(k) \vee (\tau_i \in lp(k) \wedge \gamma < A_k) \\ W_{i,k}^{n_3} & otherwise \end{cases}, \tag{10}$$

where $W_{i,k}^{n_1}$, $W_{i,k}^{n_2}$, $W_{i,k}^{n_3}$ are defined in Equations (1), (2) and (3) respectively.

4.2.4 Upper Bound on $W_{i,k}^c$ for FP_{np}

We compute the upper bound on $W_{i,k}^c$ by three cases: (1) $i = k$, (2) $\tau_i \in lp(k) \wedge S_k \ge C_i^*$, (3) the remaining cases.

1) $i = k$. The worst case of $W_{i,k}^c$ occurs as it does for EDF_{np}, and therefore $W_{i,k}^c$ is computed by Equation (5).
2) $\tau_i \in lp(k) \wedge S_k \ge C_i^*$. The worst case of $W_{i,j}^c$ occurs when one of τ_i's job is released at $r_k^j - 1$, as shown in case (b) of Fig. 7. We can compute $W_{i,j}^c$ by Equation (7).
3) The remaining cases, i.e., $\tau_i \in hp(k)$ or $\tau_i \in lp(k) \wedge C_i^* > S_k$. The worst-case workload of τ_i is generated when one of τ_i's instances is released at time instance $s_k^j - C_i^*$. Such a situation is depicted in Fig. 8. In this case, we can compute $W_{i,j}^c$ by Equation (8).

By the above discussion, we compute $W_{i,j}^c$ by

$$W_{i,k}^c = \begin{cases} W_{i,k}^{c_1} & i = k \\ W_{i,k}^{c_3} & \tau_i \in lp(k) \wedge S_k \ge C_i^* \\ W_{i,k}^{c_4} & otherwise \end{cases}, \tag{11}$$

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

4.2.5 Upper Bound on I_k^{pre}

By the definition of o_k^j, at least one core is idle at o_k^j, therefore at most $m - 1$ tasks have carry-in jobs. The task set τ can be partitioned into two subsets τ^c and τ^n that include tasks with carry-in jobs and tasks without carry-in jobs, respectively. Now we define Ω_k as the maximal value of the sum of all tasks' workloads in $[o_k^j, l_k^j]$ among all possible cases

$$\begin{aligned} \Omega_k &= \max \sum_{\tau_i \in \tau} W_{i,k} \\ &= \max_{(\tau^n, \tau^c) \in \tau} \left(\sum_{\tau_i \in \tau^n} W_{i,k}^n + \sum_{\tau_i \in \tau^c} W_{i,k}^c \right), \end{aligned} \tag{12}$$

where τ^n and τ^c satisfy $\tau^n \cup \tau^c = \tau$, $\tau^n \cap \tau^c = \varnothing$ and $|\tau^c| \leq m-1$.

By taking the maximum over the task set, Ω_k describes an upper bound on the total worst-case workload in $[o_k^j, l_k^j]$. The complexity to compute Ω_k is $\mathcal{O}(n)$, as explained in [18].

Since both EDF_{np} and FP_{np} are work-conserving, the processor-contention interference exhibited by τ_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 $\bar{I}_k^{sc}$

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 τ_k and τ_i. Let τ_k be the interfered and τ_i 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 τ_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 a 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 τ_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 τ_k is evicted by τ_i and consequently causing reload overhead for τ_k. A cache line that may cause cache interference for τ_k needs to satisfy at least two conditions:

(i) access to that cache line will result in a cache hit at the LLC cache in WCET analysis of τ_k,
(ii) the cache line may be used by τ_i.

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 τ_k, while the second condition indicates that accessing to CB_i by τ_i may interfere with τ_k. Furthermore, cache interference occurs only if τ_k accesses memory blocks in HB_k and τ_i accesses memory blocks in CB_i concurrently, and those memory blocks have the same cache index.

We use $I_{i,k}^{sc}$ to represent the upper bound on the shared cache interference imposed on τ_k by only one job execution of τ_i.

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 idx that maps a memory address to the cache line index at the LLC cache as follows:

$$\hat{m}_{k,u} = \{m_{k,x} \in HB_k | idx(m_{k,x}) = u\}, (0 \leq u < N, u \in \mathbb{N}).$$

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

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

Let $N_{k,u}$ represent the number of hit accesses to the uth cache line by τ_k without cache interference. $N_{k,u}$ equals to the total number of access to the HBs mapping to the kth cache line

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

Similarly, we divide CB_i into N subsets by

$$\hat{e}_{i,u} = \{m_{i,x} \in CB_i | idx(m_{i,x}) = u\}, (0 \leq u < N, u \in \mathbb{N}).$$

The number of accesses to the kth cache line by τ_i is bounded by

$$N_{i,u} = \sum_{x=1}^{q} n_{i,x} \chi_{\hat{e}_{i,u}}(m_{i,x}),$$

Cache interference can only happen among memory blocks that are in the same subset that maps to the same cache line. For the uth cache line, τ_k can be interfered at most $N_{k,u}$ times and τ_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 τ_k

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

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

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

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

Proof. The lemma holds as discussed above. □

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

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

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

Proof. For $N_{i,k}$ jobs of τ_i, the total number of accesses to each memory block $m_{i,x}$ is bounded by $N_{i,k} n_{i,x}$. Thus, the execution of $N_{i,k}$ jobs of τ_i accesses the kth 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 τ_k from $N_{i,k}$ jobs of τ_i is $\sum_{u=0}^{N-1} min(N_{i,k}N_{i,u}, N_{k,u}) C_{miss}$

$$\begin{aligned} N_{i,k} I_{i,k}^{sc} &= N_{i,k} \sum_{u=0}^{N-1} min(N_{i,u}, N_{k,u}) C_{miss} \\ &= \sum_{u=0}^{N-1} min(N_{i,k}N_{i,u}, N_{i,k}N_{k,u}) C_{miss} \\ &\geq \sum_{u=0}^{N-1} min(N_{i,k}N_{i,u}, N_{k,u}) C_{miss}. \end{aligned}$$

□

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 τ_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 τ_i is schedulable without cache interference.

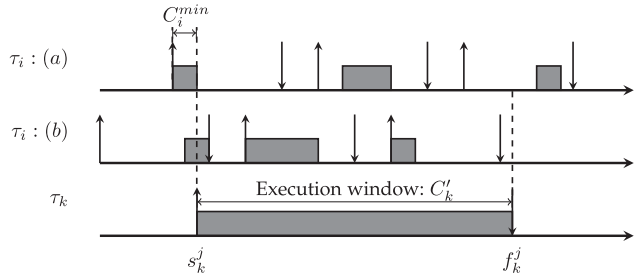

Fig. 10. Situations where τ_i interferes τ_k with the most and least number of jobs.

If $N_{i,k}$ jobs of τ_i are executing concurrently with τ_k, the cache interference that τ_i causes on τ_k is bounded by $N_{i,k} I_{i,k}^{sc}$ 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 τ_k is bounded by the sum of the contributions of all other tasks $\tau_i (i \neq k)$ in the task set τ. Thus, the objective function of the IP formulation is

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

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 jth job of τ_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 τ_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_i^{min} as the smallest execution time of τ_i, often called Best-Case Execution Time (BCET), we have the following constraint:

$$\forall i \neq k, \left\lfloor \frac{\max(0, C'_k - T_i + C_i^{min})}{T_i} \right\rfloor + \xi_i \leq N_{i,k}, \tag{14}$$

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

The term ξ_i indicates whether the last job of τ_i released within the EW will interfere with τ_k, since the last released job should start its execution C_i^{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 τ_i finishes just after the start of the EW and the last interfering job of τ_i starts to execute at the time when it is released. Such a situation is depicted as case (b) in Fig. 10. Thus, we have the second constraint on $N_{i,k}$

$$\forall i \neq k, \; N_{i,k} \leq 1 + \left\lceil \frac{\max(0, C'_k - T_i + D_i)}{T_i} \right\rceil. \tag{15}$$

If $N_{i,k} > 2$, the first and last interfering jobs of τ_i may occupy almost 0 computation capacity in the EW. Let J_i^j be

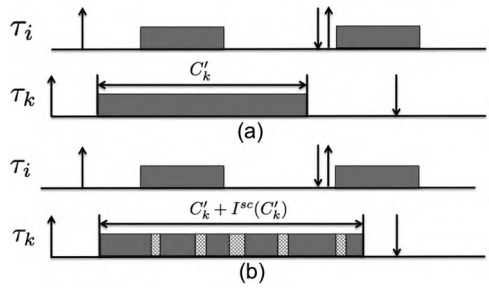

Fig. 11. More cache interference if τ_k executes for a longer time.

such a job among the remaining $N_{i,k} - 2$ interfering jobs of τ_i between the first and the last ones. Both release time r_i^j and deadline d_i^j of J_i^j are within the EW of τ_k.

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

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

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 τ_k. Due to the limited computation capacity of the processor, the total execution of the tasks that may interfere with τ_k within the EW can not exceed $(m-1)C'_k$. Hence, we have the third constraint

$$\sum_{i \neq k} \max(0, N_{i,k} - 2) C_i \leq (m-1)\ C'_k. \tag{16}$$

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 τ_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., $\bar{I}_i^{sc} = 0$.

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 $\mathcal{O}(n64^n \ln 4n)$ [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 τ_k executes for C'_k, only one job of τ_i interferes with τ_k. In Fig. 11b, if the job of τ_k executes for a larger execution time, say $C'_k + I^{sc}(C'_k)$, two jobs of τ_i could possibly interfere with τ_k, which potentially may increase the cache interference exhibited by τ_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 $\bar{I}_k^{sc} = I^{sc}(C_k^*)$.*

Proof. If $C_k^* = C_k + I^{sc}(C_k^*)$, then $I^{sc}(C_k^*) = I^{sc}(C_k + I^{sc}(C_k^*))$. According to Lemma 8, given an execution window of τ_k that is no more than $C_k + I^{sc}(C_k^*)$, the cache interference exhibited by τ_k is not larger than $I^{sc}(C_k^*)$. Therefore, $I^{sc}(C_k^*)$ is the upper bound on cache interference for τ_k. By definition, $\bar{I}_k^{sc} = I^{sc}(C_k^*)$. □

We now derive the iterative algorithm, called $CacheInterference(\tau, m)$ to compute an upper bound on cache interference for each task $\tau_k \in \tau$:

- 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 τ_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 τ_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 τ_k unschedulable.
- The previous steps are repeated for every task in τ.

A more formal version of the $CacheInterference(\tau, m)$ 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 τ_k and C^* which includes the upper bounds on the execution length C_k^* for each τ_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.

Pseudocode 1. CacheInterference(τ, m)

```
Input: Task parameters, number of cores: m
I* ← empty list, used to store I^sc(C*_k) for each task
C* ← empty list, used to store C*_k for each task
for all τ_k ∈ τ do
    update ← true, I^old_k ← 0, I^new_k ← 0
    C'_k ← C_k
    while update do
        I^old_k    I^new_k
        I^new_k    Solution of IP with C'_k as the EW
        C'_k = C_k + I^new_k
        if I^new_k == I^old_k or C'_k ≥ D_k then
            update ← false
        end if
    end while
    Add I^new_k to I*
    Add C'_k to C*
end for
return I*, C*
```

We denote $U(\tau_i)$ as task τ_i's utilization taking shared cache interference into account, $U(\tau_i)$ is defined by

$$U_i = \frac{C_i^*}{T_i}.$$

The utilization of taskset τ, denoted by $U(\tau)$, is defined by

$$U(\tau) = \sum_{\tau_i \in \tau} U_i = \sum_{\tau_i \in \tau} \frac{C_i^*}{T_i}.$$

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

$$\Delta_{C_i^*}^{m-1} = \sum_{the\ (m-1)\ largest} C_i^*.$$

For task τ_k, we also define a constant L_k by

$$L_k = \frac{\sum_{\tau_i \in \tau} C_i^* + \Delta_{C_i^*}^{m-1}}{m - U(\tau)} - S_k. \tag{17}$$

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

Theorem 2. *A task set τ is schedulable with the EDF_{np} or FP_{np} scheduling policy on a multicore platform composed of m identical cores with shared caches if for each task $\tau_k \in \tau$ and $0 \le A_k \le L_k$,*

(1) $\exists\, C_k^* \ge C_k$ *such that* $C_k^* = C_k + I^{sc}(C_k^*)$,
(2) $\frac{\Omega_k}{m} + C_k^* < D_k + A_k$.

Proof. From (1), $\bar{I}_k^{sc}$ is bounded and $\bar{I}_k^{sc} = I^{sc}(C_k^*)$ according to Lemma 9.

From Lemma 4, $\bar{I}_k^{pre} = \frac{\Omega_k(C^*)}{m}$.

$\forall A_k \ge 0$, if $\frac{\Omega_k}{m} + C_k^* = \frac{\Omega_k}{m} + C_k + I^{sc}(C_k^*) < A_k + D_k$ then $\bar{I}_k^{pre} + C_k + \bar{I}_k^{sc} < A_k + D_k$. Theorem 2 follows from Theorem 1.

We further prove that if condition (2) is to be violated for any A_k, then it must also be violated for some $A_k \le L_k$.

$W_{i,k}^n$ can be bounded by considering the number of body jobs to be $\left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor$ and the contribution of the carry-out to be C_i^*, so

$$W_{i,k}^n \le \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor C_i^* + C_i^* \le \frac{A_k + S_k}{T_i} C_i^* + C_i^* = (A_k + S_k)U_i + C_i^*.$$

Similarly, $W_{i,k}^c$ can be bounded by considering the contribution of both the carry-in and the carry-out are C_i^*

$$W_{i,k}^c \le \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor C_i^* + 2C_i^* \le (A_k + S_k)U_i + 2C_i^*.$$

From Equation (12)

$$\begin{aligned}\Omega_k &= \max_{(\tau^n,\tau^c)\in\tau} \left(\sum_{\tau_i \in \tau^n} W_{i,k}^n + \sum_{\tau_i \in \tau^c} W_{i,k}^c \right) \\ &\le (A_k + S_k) \sum_{\tau_i \in \tau} U_i + \sum_{\tau_i \in \tau} C_i^* + \Delta_{C_i^*}^{m-1} \\ &= (A_k + S_k)U(\tau) + \sum_{\tau_i \in \tau} C_i^* + \Delta_{C_i^*}^{m-1}.\end{aligned}$$

If condition (2) is to be violated for any A_k, then $\exists A_k, \frac{\Omega_k}{m} + C_k^* \ge D_k + A_k$

$$\begin{aligned}&\Longrightarrow \Omega_k \ge m(D_k + A_k - C_k^*) \\ &\Longrightarrow (A_k + S_k)U(\tau) + \sum_{\tau_i \in \tau} C_i^* + \Delta_{C_i^*}^{m-1} \ge m(S_k + A_k).\end{aligned}$$

Solve the above inequality for A_k, we have

$$A_k \le \frac{\sum_{\tau_i \in \tau} C_i^* + \Delta_{C_i^*}^{m-1}}{m - U(\tau)} - S_k = L_k.$$

This tells us the range of A_k that should be tested. □

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

Pseudocode 2. CheckSchedulability(τ, m)

```
Input: Task parameters, number of cores: m
I*, C* ← CacheInterference(τ, m)
for all τ_k ∈ τ do
    calculate L_k by Equation (17)
    for all A_k ∈ [0, L_K] do
        Ω_k ← calculation of Equation (12) using C*, A_k
        if Ω_k/m + C*_k ≥ D_k + A_k then
            return Unschedulable
        end if
    end for
end for
return Schedulable
```

Computational Complexity. Let n represent the number of tasks in the task-set. For τ_k, let I_k^{min} be the smallest difference between cache interference caused by one job of τ_i and τ_j, i.e., $I_k^{min} = \min_{i,j}(I_{i,k}^{sc} - I_{j,k}^{sc})$, the iterative algorithm takes at most $\eta = \max_k \frac{(D_k - C_k)}{I_k^{min}}$ iterations to terminate since C'_k either stays the same or increases at least with I_k^{min} in each iteration. Thus, the complexity of the iterative algorithm to compute the upper bound on cache interference is $\mathcal{O}(\eta n^2 64^n ln4n)$. The complexity of computing L_k, Ω_k is polynomial. Therefore, the complexity to perform the schedulability test is $\mathcal{O}(\eta n^2 64^n ln4n)$.

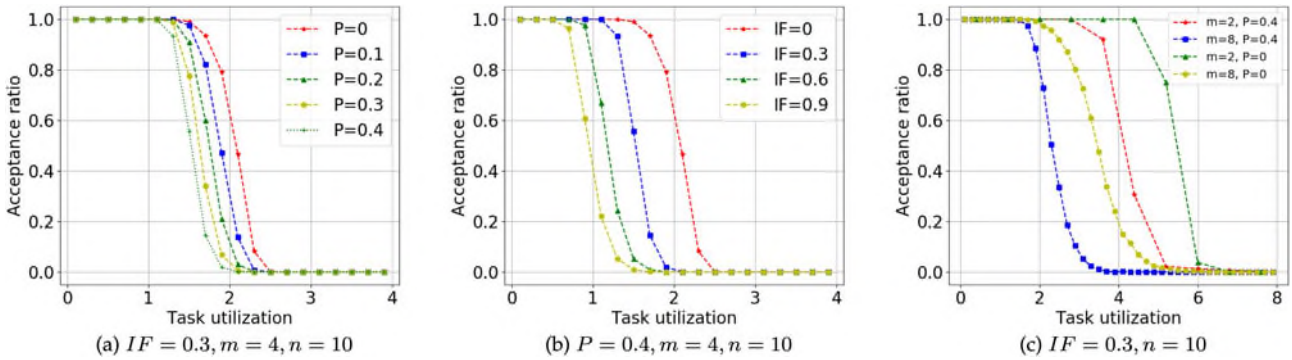

Fig. 12. Acceptance ratio of EDF_{np} when varying cache interference factor: IF, probability: P and number of cores: m.

6 Experiments

In this section, we systematically generate synthetic workloads to evaluate the performance of the proposed schedulability test for EDF_{np} and FP_{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 EDF_{np} against FP_{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 tasksets 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 Randfixedsum [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 τ_k, T_k is uniformly distributed over the interval $[100, 200]$. The WCET of τ_k is derived by $C_k = T_k \times U_k$. We consider an implicit deadline task system, which implies that $D_i = T_i$.

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

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 EDF_{np} and FP_{np}, respectively.

Fixing $m = 4,\ n = 10, IF = 0.3$, Figs. 12a and 13a illustrate the acceptance ratio with different P for EDF_{np} and FP_{np}, respectively. With the same U_{tot}, the acceptance ratio for both EDF_{np} and FP_{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 EDF_{np} is higher than FP_{np} when $U_{tot} \in [1.1, 2.5]$. For example, when $P = 0.2, IF = 0.3$ and $U_{tot} = 1.7$, 60.1 percent of tasksets are schedulable by EDF_{np}, while FP_{np} schedules 50.45 percent of the generated tasksets.

Figs. 12b and 13b show the acceptance ratio achieved by EDF_{np} and FP_{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 EDF_{np} is better than FP_{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 percent of tasksets are schedulable by EDF_{np}, while FP_{np} schedules 59.3 percent of the generated tasksets.

Figs. 12c and 13c illustrate the acceptance ratio with respect to the number of cores for EDF_{np} and FP_{np}, respectively. In

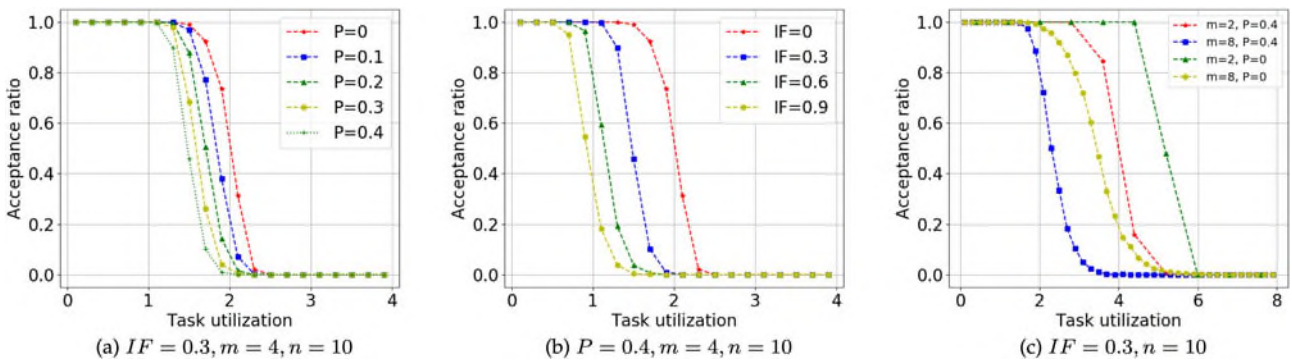

Fig. 13. Acceptance ratio of FP_{np} when varying cache interference factor: IF, probability: P and number of cores: m.

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 $\frac{U_{tot} \times 8}{m}$ 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 linear 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

[1] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 80–89.

[2] E. Berg, H. Zeffer, and E. Hagersten, "A statistical multiprocessor cache model," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2006, pp. 89–99.

[3] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proc. 45th ACM/IEEE Des. Autom. Conf.*, 2008, pp. 300–303.

[4] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *Proc. Real-Time Syst. Symp.*, 2008, pp. 456–466.

[5] J. Xiao, S. Altmeyer, and A. Pimentel, "Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches," in *Proc. IEEE Real-Time Syst. Symp.*, 2017, pp. 199–208.

[6] R. Wilhelm *et al.*, "The worst-case execution-time problem—Overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

[7] W. Zhang and J. Yan, "Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches," in *Proc. 15th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2009, pp. 455–463.

[8] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *Proc. 30th IEEE Real-Time Syst. Symp.*, 2009, pp. 68–77.

[9] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra, "Timing analysis of concurrent programs running on shared cache multi-cores," *Real-Time Syst.*, vol. 48, no. 6, pp. 638–680, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11241-012-9160-2

[10] G. Gracioli and A. A. Fröhlich, "An experimental evaluation of the cache partitioning impact on multicore real-time schedulers," in *Proc. IEEE 19th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2013, pp. 72–81.

[11] J. Liedtke, H. Hartig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Proc. 3rd IEEE Real-Time Technol. Appl. Symp.*, 1997, pp. 213–224.

[12] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 157–167.

[13] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp.*, 2013, pp. 45–54.

[14] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 1–14.

[15] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller, "Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems," in *Proc. 24th Euromicro Conf. Real-Time Syst.*, 2012, pp. 331–340.

[16] M. Xu, L. Thi, X. Phan, H. Choi, and I. Lee, "vCAT: Dynamic cache management using cat virtualization," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2017, pp. 211–222.

[17] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Proc. 27th IEEE Int. Real-Time Syst. Symp.*, 2006, pp. 101–110.

[18] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *Proc. 28th IEEE Int. Real-Time Syst. Symp.*, 2007, pp. 119–128. [Online]. Available: http://dx.doi.org/10.1109/RTSS.2007.48

[19] N. Guan, G. Yu, W. Yi, Q. Deng, and Z. Gu, "New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms," in *Proc. Real-Time Syst. Symp.*, 2008, pp. 137–146. [Online]. Available: doi.ieeecomputersociety.org/10.1109/RTSS.2008.17

[20] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 4, pp. 553–566, Apr. 2009.

[21] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with EDF scheduling," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1250–1258, Sep. 2009.

[22] J. Lee, K. G. Shin, I. Shin, and A. Easwaran, "Composition of schedulability analyses for real-time multiprocessor systems," *IEEE Trans. Comput.*, vol. 64, no. 4, pp. 941–954, Apr. 2015.

[23] D. Liu *et al.*, "Scheduling analysis of imprecise mixed-criticality real-time tasks," *IEEE Trans. Comput.*, vol. 67, no. 7, pp. 975–991, Jul. 2018.

[24] L. Sha *et al.*, "Real time scheduling theory: A historical perspective," *Real-Time Syst.*, vol. 28, no. 2/3, pp. 101–155, Nov. 2004. [Online]. Available: https://doi.org/10.1023/B:TIME.0000045315.61234.1e

[25] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/1978802.1978814

[26] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proc. 7th ACM Int. Conf. Embedded Softw.*, 2009, pp. 245–254.

[27] M. Xu, L. T. X. Phan, H. Y. Choi, and I. Lee, "Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2016, pp. 1–12.
[28] M. Xu *et al.*, "Holistic resource allocation for multicore real-time systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 345–356.
[29] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *Proc. 23rd Int. Conf. Real Time Netw. Syst.*, 2015, pp. 129–138.
[30] T. P. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *Proc. 24th IEEE Real-Time Syst. Symp.*, 2003, pp. 120–129.
[31] K. L. Clarkson, "Las Vegas algorithms for linear and integer programming when the dimension is small," *J. ACM*, vol. 42, pp. 488–499, 1995.
[32] R. Stafford, "Random vectors with fixed sum," 2006. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/9700

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 Modelling Practice and Theory* as well as the *Springers 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.