Schedulability Analysis of Non-preemptive Real-time Scheduling for Multicore Processors with Shared Caches

Jun Xiao*, Sebastian Altmeyer† and Andy Pimentel‡
Informatics Institute, University of Amsterdam, Amsterdam, Netherlands
Email: *J.Xiao@uva.nl, †S.J.Altmeyer@uva.nl, ‡A.D.Pimentel@uva.nl

***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. To the best of our knowledge, this is the first work that addresses the schedulability problem with inter-core cache interference. 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.**

I. 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 [13], 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 [4].

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 [19], 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, task's WCET does not account for shared cache interference. [12] presents such an approach to derive a task's WCET without considering shared cache interference. We propose a novel schedulability analysis of non-preemptive real-time scheduling for multicore systems with shared caches. Intra-core cache interference is avoided since no preemption is possible during task execution. We therefore focus on inter-core cache interference and specially conflict misses that caused by limited cache associativity. The main contributions of this work are:

- An integer programming formulation is constructed to calculate the upper bound on the cache interference exhibited by a task within a given execution window;
- An iterative algorithm is presented to obtain the upper bound on inter-core cache interference a task may exhibit during its job executions;
- A new schedulability condition is derived by integrating the upper bound on inter-core cache interference into the schedulability analysis;
- A set of experiments are performed using the proposed schedulability analysis to investigate the effects of inter-core cache interference for a range of different tasksets.

The rest of the paper is organized as follows. Section II gives an overview of the related work. The system model is described in Section III. Section IV describes the proposed schedulability analysis, where we also detail the computation of processor-

contention and inter-core cache interferences applied in the analysis. Section V presents an iterative computation to obtain the upper bound of inter-core cache interferences. Section VI presents the experimental results, after which Section VII concludes the paper.

II. Related work

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 [21]. 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 [23], [11], [15]. 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.

Since shared caches introduce the difficulty into 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 [8]. The most common software-based cache partitioning technique is page coloring [16], [20]. By exploiting the virtual to physical page address translations present in virtual memory systems at OS-level, page addresses are mapped to pre-defined cache regions to avoid the overlap of cache spaces. Hardware-based cache partitioning is achieved using a cache locking mechanism [6], [19], [17], which prevents cache lines from being evicted during program execution. The main drawback of cache locking is that it requires specific hardware support that is not available in many commercial processors. With shared cache partitioning techniques, one can apply existing analyses to derive the upper bounds of a task's WCET assuming that no cache interference can occur between tasks simultaneously running on different cores. In that case, it is safe to use the derived WCETs in the schedulability analysis.

Although the schedulability analysis of global multiprocessor scheduling has been intensively studied [14], [5], [3], few works addressing schedulability analysis for multicores with shared caches are [9], [22], where cache space isolation is deployed. In addition to regular temporal constraints, cache constraints due to cache space isolation are added in the schedulability analysis. They propose a linear programming formulation to perform the schedulability test and an over approximation of this formulation to improve the scalability of the test. However, this solution is not applicable to our problem since our system architecture does not deploy any cache isolation techniques.

Our work also differs from other approaches to the timing verification of multicore systems [1] 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 the first work that integrates inter-core cache interferences into schedulability analysis.

III. System Model

A. Task Model

We consider a set τ of n periodic or sporadic real-time tasks τ_1, τ_2, ... τ_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 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.

B. 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 different independent caches, including 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 noninclusive with respect to the private caches ($L1$), and that *LLC* caches are direct-mapped caches. We believe this work can be extended to set-associative LRU caches, and we plan to do so as future work.

In this work, we only consider instruction caches since we adopt the approach in [12], which only accounts for instruction cache, to derive WCET.

C. NP-FP Scheduler

In this paper, we focus on non-preemptive global scheduling, thus 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. For simplicity reasons, we will take the Non-Preemptive Fixed Priority (NP-FP) global scheduling as the example in this paper. We will extend our work to other non-preemptive schedulers such as a global Non-Preemptive Earliest Deadline First (NP-EDF) scheduler.

To use NP-FP scheduling, a priority P_k is assigned to each task τ_k $(k = 1, 2, ...n)$. 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.

The NP-FP scheduling algorithm is 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.

IV. 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.

A. Overview

We first analyze the execution of one job J_k^j of a task τ_k. The time interval $[r_k^j, d_k^j]$ is called a problem window [3].

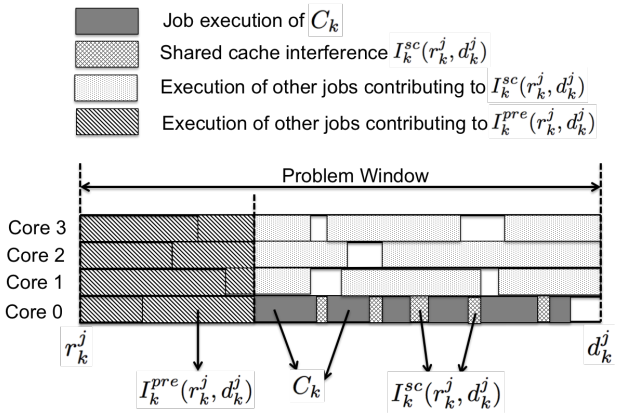

Figure 1: Overview of the new schedulability analysis that accounts for cache interference.

As shown in Figure 1, a task τ_k exhibits two kinds of interferences during the execution of a job. The first interference is $I_k^{pre}(r_k^j, d_k^j)$, denoting the cumulative length of all intervals over the problem window in which τ_k is ready to execute but cannot proceed due to unavailability of cores as they are executing other jobs. We define the interference $I_{i,k}^{pre}(r_k^j, d_k^j)$ of a task τ_i on a task τ_k over the interval $[r_k^j, d_k^j)$ as the cumulative length of all intervals in which τ_k is ready to execute, and τ_i is executing while τ_k is not. The second type of interference is the cumulative length of all extra execution delays caused by shared cache interference due to conflict accesses from all other tasks running concurrently on other cores, denoted as $I_k^{sc}(r_k^j, d_k^j)$. We also define the interference $I_{i,k}^{sc}(r_k^j, d_k^j)$ as the cumulative length of all extra execution delays of τ_k over the problem window caused by conflict shared cache accesses between task τ_i and task τ_k.

It is clear that for a job to meet its deadline, the sum of the two interferences a task exhibits in the problem window plus the task's WCET C_k must be less than the length of the problem window, which is D_k. For a task to be schedulable, this condition must hold for all its jobs.

We define the worst-case interference for task τ_k as:

$$\begin{aligned}\bar{I}_k &= \max_j(I_k^{pre}(r_k^j, d_k^j) + I_k^{sc}(r_k^j, d_k^j)) \\ &= I_k^{pre}(r_k^{p*}, d_k^{p*}) + I_k^{sc}(r_k^{p*}, d_k^{p*})\end{aligned}$$

where $p*$ is the job instance in which the sum of the two interferences is maximal.

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

Theorem 1. *A task set τ is schedulable with a NP-FP scheduling policy on a multicore processor composed of m identical cores with shared caches if and only if for each task $\tau_k \in \tau$*

$$\bar{I}_k + C_k < D_k$$

The necessary and sufficient schedulability condition expressed by Theorem 1 cannot be used to check if a task set is schedulable without knowing how to compute the interference terms $\bar{I}_k$. Unfortunately, we are not aware of any method to compute $\bar{I}_k$ starting from the given task parameters. To sidestep this problem, we will use an upper bound on each of the interferences.

We define the worst-case processor-contention interference $\bar{I}_k^{pre}(r_k^j, d_k^j)$ and worst-case shared cache interference $\bar{I}_k^{sc}(r_k^j, d_k^j)$ for task τ_k as:

$$\bar{I}_k^{pre} = \max_j(I_k^{pre}(r_k^j, d_k^j)) = I_k^{pre}(r_k^{q*}, d_k^{q*})$$

where $q*$ is the job instance in which the processor-contention interference is maximal and

$$\bar{I}_k^{sc} = \max_j(I_k^{sc}(r_k^j, d_k^j)) = I_k^{sc}(r_k^{s*}, d_k^{s*})$$

where $s*$ is the job instance in which the cache interference is maximal, respectively.

Theorem 2. *A task set τ is schedulable with a NP-FP scheduling policy on a multicore processor composed of m identical cores with shared caches if for each task $\tau_k \in \tau$*

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

Proof.

$$\begin{aligned}\bar{I}_k &= \max_j(I_k^{pre}(r_k^j, d_k^j) + I_k^{sc}(r_k^j, d_k^j)) \\ &\leq \max_j(I_k^{pre}(r_k^j, d_k^j)) + \max_j(I_k^{sc}(r_k^j, d_k^j)) \\ &= \bar{I}_k^{pre} + \bar{I}_k^{sc}\end{aligned}$$

if $\bar{I}_k^{pre} + C_k + \bar{I}_k^{sc} < D_k$, then $\bar{I}_k + C_k < D_k$. The Theorem follows from Theorem 1. □

B. Computing an upper bound of $\bar{I}_k^{pre}$

The workload $W_i(D_k)$ of a task τ_i in the problem window $[r_k^j, d_k^j)$ of length D_k is the time task τ_i executes during interval $[r_k^j, d_k^j)$, according to a given scheduling policy.

Lemma 1. *The processor-contention interference that a task τ_i causes on a task τ_k in the problem window of τ_k is never greater than the workload of τ_i in the problem window.*

$$\forall i, k, j \quad I_{i,k}^{pre}(r_k^j, d_k^j) \leq W_i(D_k)$$

Lemma 1 is obvious, since $W_i(D_k)$ is an upper bound on the execution of τ_i in the problem window.

Since it is difficult to compute the exact value of $\bar{I}_k^{pre}$, we will compute the upper bound of the worst-case workload by each task in the problem window, and use the sum of each task's workload to derive an upper bound on $\bar{I}_k^{pre}$.

As shown in Figure 2, the upper bound of the worst-case workload can be calculated by categorizing each job of τ_i in the problem window into one of the three types [2]:

carry-in job: a job with its release time earlier than r_k^j, but with its deadline in the problem window;

body job: a job with both its release time and its deadline in the problem window;

carry-out job: a job with its release time in the problem window, but with its deadline later than d_k^j.

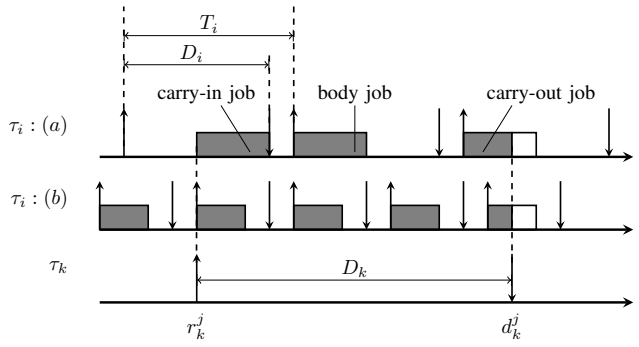

Figure 2: Three types of contribution jobs and problem window. Case (a) and (b) shows the densest possible packing of jobs of τ_i if τ_i has a carry-in job and has no carry-in jobs, respectively.

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 $W_i^{nc}(D_k)$ to denote an upper bound of τ_i's workload in the problem window if τ_i has no carry-in job, and use $W_i^{ci}(D_k)$ to denote an upper bound of τ_i's workload if τ_i has a carry-in job. Following the approach in [10], we compute $W_i^{nc}(D_k)$ and $W_i^{ci}(D_k)$ as follows:

- Computing $W_i^{nc}(D_k)$
  For $\tau_i \in hp(k)$, the worst-case workload of task τ_i occurs when a job of τ_i arrives at exactly the start of the problem window, as shown in case (b) in Figure 2. The next jobs of τ_i are then released periodically every T_i time units. Therefore the number of body jobs of τ_i that contribute with C_i to the workload in the problem window is at most $\left\lfloor \frac{D_k}{T_i} \right\rfloor$. The contribution of the carry-out job can then be bounded by $\min(D_k \mod T_i, C_i)$.
  τ_i's workload in the problem window is 0 for $\tau_i \in lep(k)$. We can compute $W_i^{nc}(D_k)$ by:

$$W_i^{nc}(D_k) = \begin{cases} \left\lfloor \frac{D_k}{T_i} \right\rfloor C_i + \omega^{nc} & \tau_i \in hp(k) \\ 0 & \tau_i \in lep(k) \end{cases} \tag{2.1}$$

  where $\omega^{nc} = \min(D_k \mod T_i, C_i)$
- Computing $W_i^{ci}(D_k)$
  If $\tau_i \in hp(k)$, the worst-case workload of τ_i is generated when r_k^j coincides with the starting time of the carry-in job of τ_i: moving the problem window backwards, the contribution of carry-in cannot increase and the carry-out can only decrease; while advancing the problem window, the carry-in will decrease and the carry-out can increase by at most the same amount. Such a situation is depicted as case (a) in Figure 2. The contribution of the carry-in job is bounded by C_i. Note that the first body job of τ_i after the carry-in jobs, is released at time $r_k^j + C_i + T_i - D_i$. The number of body jobs that contribute to τ_i's workload is $N_i(D_k) = \left\lfloor \frac{\max(0, D_k - C_i - T_i + D_i)}{T_i} \right\rfloor$. The contribution of the carry-out job can then be bounded by $\min(C_i, \max\ (0, D_k - C_i - T_i + D_i) \mod T_i)$.
  For $\tau_i \in lep(k)$, only the carry in job of τ_i starting execution before r_k^j can contribute to the workload in the problem window.
  Thus, we compute $W_i^{ci}(D_k)$ by:

$$W_i^{ci}(D_k) = \begin{cases} (1 + N_i(D_k))C_i + \omega^{ci} & \tau_i \in hp(k) \\ \min(D_k, C_i) & \tau_i \in lep(k) \end{cases} \tag{2.2}$$

  where

$$\omega^{ci} = \min(C_i, \max\ (0, D_k - C_i - T_i + D_i) \mod T_i).$$

Lemma 2. *If tasks are scheduled with a NP-FP scheduling policy on a multicore processor composed of m identical cores, at most m tasks have carry-in jobs.*

Proof. See Lemma 5.2. in [10]. □

The task set τ can be partitioned into two subsets τ^{nc} and τ^{ci} that include tasks with carry-in jobs and tasks without carry-in jobs in the problem window, respectively. According to Lemma 2, τ^{ci} has at most m tasks. Now we define Ω_k as the maximal value of the sum of all tasks' workloads (other

than τ_k' workload) in the problem window of τ_k among all possible cases:

$$\begin{aligned}\Omega_k &= \max \sum_{i \neq k} W_i(D_k) \\ &= \max_{(\tau^{nc},\tau^{ci}) \in \tau} \Big(\sum_{\tau_i \in \tau^{nc}} W_i^{nc}(D_k) + \sum_{\tau_i \in \tau^{ci}} W_i^{ci}(D_k) \Big)\end{aligned} \tag{2.3}$$

where τ^{nc} and τ^{ci} satisfy $\tau^{nc} \cup \tau^{ci} = \tau \backslash \{\tau_k\}$, $\tau^{nc} \cap \tau^{ci} = \varnothing$ and $|\tau^{ci}| \leq m$.

By taking the maximum over the task set, Ω_k describes an upper bound on the total worst-case workload (other than τ_k' workload) in the problem window of τ_k. The complexity to compute Ω_k is $\mathcal{O}(n)$, as explained in [3].

Replacing C_i. The computation of $W_i^{nc}(D_k)$, $W_i^{ci}(D_k)$, Ω_k depends on C_i. We denote the value Ω_k as $\Omega_k(C)$ when C_i is used in Equation (2.1) and (2.2). Fixing parameters other than C_i in Equation (2.1) and (2.2), $W_i^{nc}(D_k)$, $W_i^{ci}(D_k)$ and the resulting Ω_k are non-decreasing with respect to C_i. In the following discussion, we will show that the actual execution time of τ_i including cache interference could be larger than C_i. Since cache interference could also contribute to the task workload, we will use C_i^* which is the sum of C_i and the upper bound on cache interference exhibited by τ_i to replace C_i in Equation (2.1) and (2.2) in order to get the correct upper bound on the worst-case workload. We denote $\Omega_k(C^*)$ as resulting value if C_i^* is used in the computation.

We are now ready to compute an upper bound of $\bar{I}_k^{pre}$.

Lemma 3. *If tasks are scheduled with a NP-FP scheduling policy on a multicore processor composed of m identical cores with shared cache,*

$$\bar{I}_k^{pre} \leq \frac{\Omega_k(C^*)}{m}$$

Proof. Since the scheduling algorithm NP-FP is work-conserving, in the time instants in which a job of τ_k is ready but not executing, each core must be occupied by a job of another task. As $I_{k,k}^{pre}(r_k^{q*}, d_k^{q*}) = 0$, we can exclude the contribution of τ_k to the interference. So

$$\bar{I}_k^{pre} \leq \frac{\sum_{i \neq k} I_{i,k}^{pre}(r_k^{q*}, d_k^{q*})}{m}.$$

By Lemma 1, the interference that a task τ_i causes on a task τ_k in the problem window is bounded its workload,

$$I_{i,k}^{pre}(r_k^{q*}, d_k^{q*}) \leq W_i(D_k).$$

So, we have

$$\begin{aligned}\bar{I}_k^{pre} \leq \frac{\sum_{i \neq k} I_{i,k}^{pre}(r_k^{q*}, d_k^{q*})}{m} &\leq \frac{\max \sum_{i \neq k} W_i(D_k)}{m} \\ &= \frac{\Omega_k(C^*)}{m}\end{aligned}$$

□

C. Computing an upper bound 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.

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 [12], 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. 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). *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).

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 [12].

We use $HB_k = \{m_{k,1}, m_{k,2}, ..., m_{k,p}\}$ to represent the set of HB for task τ_k and use $n_{k,x}$ $(x = 1, 2, ..., p)$ to denote the number of $m_{k,x}$'s accesses that are classified as a AH at the LLC cache. Similarly, we define $CB_i = \{m_{i,1}, m_{i,2}, ..., m_{i,q}\}$ 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 a A or U at the LLC cache. Note that HB_k and CB_i includes 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 indices 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 u-th cache line by τ_k without cache interference. $N_{k,u}$ equals to the total number of access to the HBs mapping to the k-th 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 k-th 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 u-th 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 miss 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 $I_{i,k}^{sc}$ satisties:

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

The computation 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.

Lemma 4. $I_{i,k}^{sc} = S(\tau_i, \tau_k) C_{miss}$.

Proof. The lemma holds as discussed above. □

Given a taskset, $I_{i,k}^{sc}$ can be computed, as shown in the proof of Lemma 4. In the following discussion, we assume $I_{i,k}^{sc}$ is known.

Lemma 4 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 jobs 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 5. *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 k-th cache line also at most $N_{i,k} N_{i,u}$ times. From the proof of Lemma 4, the upper bound of the total cache interference exhibited by τ_k from $N_{i,k}$ jobs of τ_i is $\sum_{u=0}^{N-1} min(N_{i,k} N_{i,u}, N_{k,u}) C_{miss}$.

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

□

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 schedulabity 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.

If $N_{i,k}$ jobs of τ_i 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 5. 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:

$$\max \sum_{i \neq k} N_{i,k} I_{i,k}^{sc}. \tag{2.4}$$

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 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 Figure 3. Denoting C_i^{min} as the smallest execution time of τ_i, often called Best-Case Execution Time (BCET), we have the following constraint:

$$\forall i \neq k, \left\lfloor \frac{\max(0, C'_k - T_i + C_i^{min})}{T_i} \right\rfloor + \xi_i \leq N_{i,k} \quad (2.5)$$

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

The term ξ_i indicates whether the last job of τ_i released within the EW that interferes with τ_k since the last released job should start its execution C_i^{min} before its relative deadline if the task is schedulable.

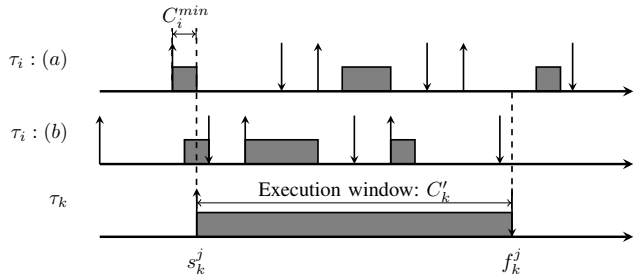

Figure 3: Situations where τ_i interferes τ_k with the most and least number of jobs.

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 Figure 3. Thus, we have the second constraint on $N_{i,k}$:

$$\forall i \neq k, \; N_{i,k} \leq 1 + \left\lceil \frac{\max(0, C'_k - T_i + D_i)}{T_i} \right\rceil \quad (2.6)$$

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 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 6. *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 (shown in Pseudocode 1): $\frac{\Omega_i(C)}{m} + C_i < D_i$, 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:

$$\sum_{i \neq k} \max(0, N_{i,k} - 2)C_i \leq (m-1)\ C'_k. \quad (2.7)$$

The objective function (2.4) together with three constraints on $N_{i,k}$ i.e. inequalities (2.5), (2.6) and (2.7) 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 is 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 (2.5) and (2.7) 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 (2.7) 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 $\mathcal{O}(4n64^n \ln 4n)$ [7].

V. 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 Figure 4.

In Figure 4 (a), if the job of τ_k executes for C'_k, only one job of τ_i interferes with τ_k. In Figure 4(b), 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 to find an upper bound on the cache interference.

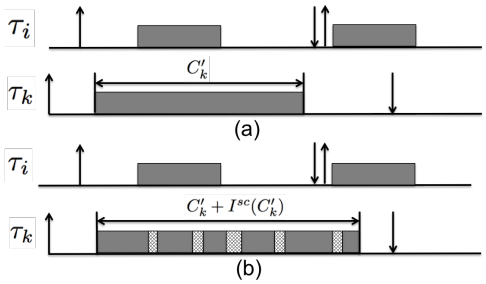

Figure 4: More cache interference if τ_k executes for a longer time.

Lemma 7. *$I^{sc}(C'_k)$ is non-decreasing with respect to C'_k*

Lemma 7 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 8. *if $\exists\ C^*_k \geq C_k$ such that $C^*_k = C_k + I^{sc}(C^*_k)$, then $I^{sc}(C^*_k)$ is the upper bound on cache interference exhibited by τ_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 7, 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. □

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

- Since the constraints of IP assume the taskset is schedulable, we first check 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 7 and one termination condition is $C'_k \geq D_k$, the termination of the algorithm is guaranteed.

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

Theorem 3. *A task set τ is schedulable with the NP-FP scheduling policy on a multicore platform composed of m identical cores with shared caches if for each task $\tau_k \in \tau$*

(1) $\exists\ C^*_k \geq C_k$ such that $C^*_k = C_k + I^{sc}(C^*_k)$,

(2) $\frac{\Omega_k(C^*)}{m} + C^*_k < D_k$.

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
pass ← true
for all τ_k ∈ τ do
    Ω_k(C) ← calculation of Equation (2.3) using C
    if Ω_k(C)/m + C_k ≥ D_k then
        pass ← false
        break
    end if
end for
if pass then
    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
end if
return I*, C*
```

Proof. From (1), $I^{sc}(C^*_k)$ is the upper bound on cache interference exhibited by τ_k according to Lemma 8. So, $I^{sc}(C^*_k) \geq \bar{I}^{sc}_k$.

From Lemma 3, $\frac{\Omega_k(C^*)}{m} \geq \bar{I}^{pre}_k$.

If $\frac{\Omega_k(C^*)}{m} + C^*_k = \frac{\Omega_k(C^*)}{m} + C_k + I^{sc}(C^*_k) < D_k$ then $\bar{I}^{pre}_k + C_k + \bar{I}^{sc}_k < D_k$. Theorem 3 follows from Theorem 2. □

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)
if I* == null then
    return Unschedulable
else
    for all τ_k ∈ τ do
        Ω_k(C*) ← calculation of Equation (2.3) using C*
        if Ω_k(C*)/m + C*_k ≥ D_k then
            return Unschedulable
        end if
    end for
end if
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 *while* loop in $CacheInterference(\tau, m)$ takes at most $\gamma = \max_k \frac{(D_k - C_k)}{I_k^{min}}$ times since C'_k either stays the same or increases at least with I_k^{min} in each iteration. Thus, the complexity of $CacheInterference(\tau)$ is $\mathcal{O}(\gamma 4n^2 64^n ln4n)$. The computational complexity of $\Omega_k(C^*)$ is $\mathcal{O}(n)$. Therefore, the complexity of $CheckSchedulability(\tau, m)$ is $\mathcal{O}(\gamma 4n^4 64^n ln4n)$.

VI. Experiments

In this section, we evaluate the performance of the proposed schedulability test in terms of acceptance ratio. More specifically, we will quantify the effects of cache interference on the schedulablity of the generated tasksets.

The experiments have been performed varying i) the number of cores m ($m = 2,\ 4$ or 8), ii) the number of tasks n ($n = 10,\ 20$, or 30) in the taskset, iii) total task utilization U_{tot} (U_{tot} from 0 to m with steps of 0.2), iv) the cache interference factor IF ($IF = 0,\ 0.3,\ 0.6$ or 0.9), and v) the probability of two tasks having cache interference on each other: P ($P = 0.1,\ 0.2,\ 0.3$ or 0.4). Given those five parameters, we have generated 200000 tasksets in each experiment. As the task generation policies may significantly affect experimental results, we give the policies used in the experiments as follows.

Task utilization generation policy. We use Randfixedsum [18] 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 $[100000, 200000]$. 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, $I_{i,k}^{sc}$ is generated as $I_{i,k}^{sc} = IF \times min(0.5C_i, 0.5C_k)$.

In each experiment, we measure the number of schedulable tasksets that pass the proposed schedulability test. The acceptance ratio is the number of schedulable tasksets devided by the total number of tasksets (200000).

Figure 5 shows the acceptance ratio for the case $IF = 0,\ 0.3,\ 0.6,\ 0.9$, when fixing $m = 8,\ n = 10,\ P = 0.1$. 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.

Figure 6 compares the acceptance ratio with different P, fixing $m = 8,\ n = 10, IF = 0.3$. With the same U_{tot}, the acceptance ratio decreases as P increases because a larger P indicates more tasks in the taskset could interference with each other, which may potentially increase the upper bound on cache interference for each task.

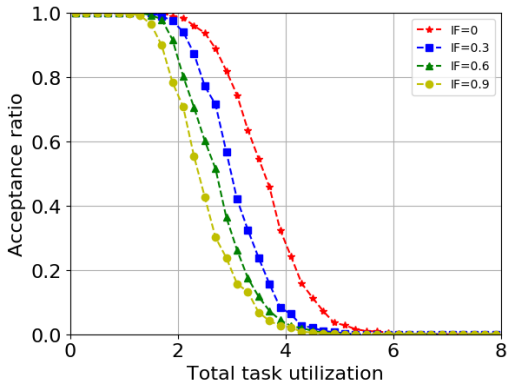

Figure 5: Acceptance ratio with different IF when $m = 8,\ n = 10,\ P = 0.1$.

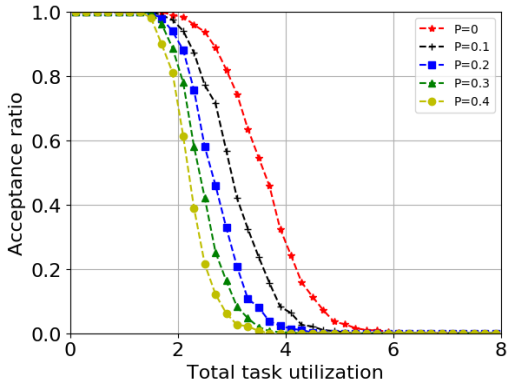

Figure 6: Acceptance ratio with different P when $m = 8,\ n = 10,\ IF = 0.3$.

Figure 7 illustrates the acceptance ratio with respect to the number of cores. The acceptance ratio for task having no cache interference is also plotted in Figure 7. 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. However, for processors with different cores, the difference in the acceptance ratio between the baseline (tasks having no cache interference, $IF = 0$) and tasks having cache interference is almost similar.

A set of experiments are performed to investigate the impact of the number of tasks in the taskset on the acceptance ratio. Figure 8 shows the acceptance ratio for different n in the taskset. It is interesting to note that when U_{tot} is less than 2, the acceptance ratio of tasksets with less tasks is slightly worse than those with more tasks. When U_{tot} is very small, U_k and C_k in a taskset with more tasks are on average smaller than those with more tasks, thus $I_{i,k}^{sc}$ is also smaller. While as U_{tot} increases, the acceptance ratio for tasksets with fewer tasks becomes better than those with more tasks. This may be due to the fact that more tasks in the taskset results in more tasks having cache interference as P is fixed.

In order to compute the average running time of the proposed

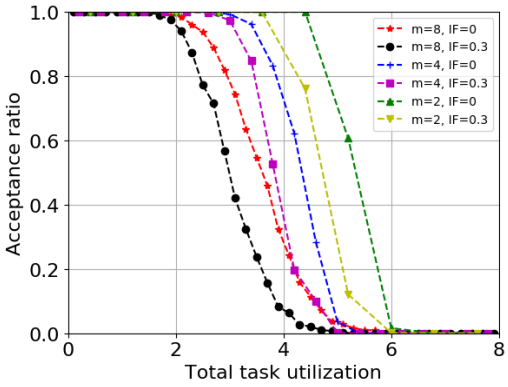

Figure 7: Acceptance ratio with different m when $IF = 0$ or 0.3, $P = 0.1$, $n = 10$.

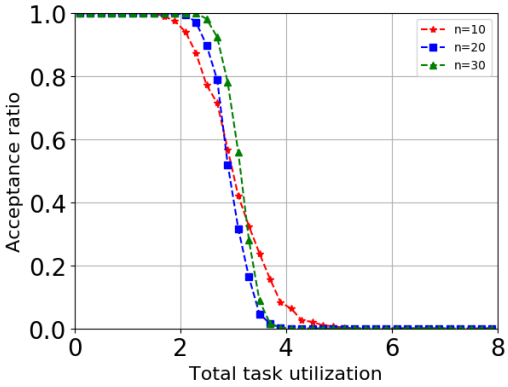

Figure 8: Acceptance ratio with different n when $IF = 0.3$, $P = 0.1$, $m = 8$.

schedulability test with different task-set scales, we measured the execution time of the schedulability test for the task-sets used in the previous experiment. The executions are conducted on a server with an 48-core AMD processor (2.1GHz). On average, it takes 0.2026 seconds to check the schedulability of the task-set consisting of 10 tasks, 0.4925 seconds for task-set with 20 tasks, while 1.0117 seconds for task-set with 30 tasks.

VII. Conclusions

In this paper, we developed a new schedulability analysis of non-preemptive real-time scheduling for multicore processors 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 derive 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. As for future work, we plan to extend our schedulability analysis to real-time multicore systems with shared caches that use preemptive task scheduling.

References

[1] 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 *23rd RTNS*, pages 129–138. ACM, 2015.

[2] T. P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *24th IEEE RTSS, 2003*, pages 120–129, Dec 2003.

[3] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *28th IEEE International RTSS*, RTSS '07, pages 119–128, Washington, DC, USA, 2007. IEEE Computer Society.

[4] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. In *2006 IEEE ISPASS*, pages 89–99, March 2006.

[5] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, April 2009.

[6] M. Caccamo, M. Cesati, R. Pellizzoni, E. Betti, R. Dudko, and R. Mancuso. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th RTAS*, RTAS '13, pages 45–54, Washington, DC, USA, 2013. IEEE Computer Society.

[7] K. L. Clarkson. Las vegas algorithms for linear and integer programming when the dimension is small. *J. ACM*, 42:488–499, 1995.

[8] G. Gracioli and A. A. Fröhlich. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 72–81, Aug 2013.

[9] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *7th ACM international conference on Embedded software*, pages 245–254. ACM, 2009.

[10] N. Guan, W. Yi, Q. Deng, Z. Gu, and G. Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536–546, 2011.

[11] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE RTSS*, pages 68–77, Dec 2009.

[12] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *2008 RTSS*, pages 456–466, Nov 2008.

[13] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *2013 25th ECRTS*, pages 80–89, July 2013.

[14] J. Lee, K. G. Shin, I. Shin, and A. Easwaran. Composition of schedulability analyses for real-time multiprocessor systems. *IEEE Transactions on Computers*, 64(4):941–954, April 2015.

[15] 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 Systems*, 48(6):638–680, 2012.

[16] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224, Jun 1997.

[17] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller. Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems. In *2012 24th ECRTS*, ECRTS '12, pages 331–340, Washington, DC, USA, 2012. IEEE Computer Society.

[18] R. Stafford. Random vectors with fixed sum http://www.mathworks.com/matlabcentral/fileexchange/9700, 2006.

[19] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *2008 45th ACM/IEEE DAC*, pages 300–303, June 2008.

[20] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *2013 25th ECRTS*, pages 157–167, July 2013.

[21] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[22] 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 *2016 IEEE RTAS*, pages 1–12, April 2016.

[23] W. Zhang and J. Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *2009 15th IEEE International Conference on RTCSA*, pages 455–463, Aug 2009.