*L*aboratoire d'*I*nformatique et d'*A*utomatique pour les *S*ystèmes

Rapport de Recherche

N^{o} 02 - 2015

03/04/2015

Complexity of scheduling real-time tasks subjected to cache-related preemption delays

Guillaume PHAVORIN, Pascal RICHARD, Claire MAIZA

ÉCOLE NATIONALE SUPÉRIEURE DE MÉCHANIQUE ET D'AÉROTECHNIQUE - UNIVERSITÉ DE POITIERS

Site du Futuroscope - Téléport 2 - 1 avenue Clément Ader
BP 40109 - 86961 FUTUROSCOPE CHASSENEUIL Cedex - FRANCE
Tél. : +33 (0) 5-49-49-80-63 - FAX : +33 (0) 5-49-49-80-64
Web : http://www.lias-lab.fr

Abstract

We consider the complexity problems of scheduling hard real-time tasks subjected to cache-related preemption delays upon uniprocessor platforms. Several schedulability analysis have been proposed in the literature to explicitly take into account preemption delays due to loss of cache affinity. But, these previous results do not study the complexity of taking scheduling decisions while taking account preemption delays and only focus on classical real-time schedulers (e.g., Rate Monotonic, Earliest Deadline First). In this paper, we focus on the computational complexity of taking scheduling decisions to meet task deadlines while minimizing cache-related preemption delay effects. We design two core cache-related scheduling problems that are the most simple NP-hard problems to cover the most largest set of intractable real-world cache-related scheduling problems. We establish several NP-hardness results in the preemptive and the non-preemptive settings. These results prove that tighter timing analysis leads in practice to harder real-time scheduling problems. These two core NP-hard scheduling problems are the following: (i) scheduling with cache-related preemption delays and (ii) scheduling with information about the cache state and the sequence of requested memory blocks for every task. We also prove for the first problem that neither fixed-task nor fixed-job priority-based scheduling algorithms can be optimal.

Contents

Complexity of scheduling real-time tasks subjected to cache-related preemption delays

Guillaume Phavorin, Pascal Richard
LIAS/Université de Poitiers
Poitiers, France
{guillaume.phavorin,pascal.richard}@univ-poitiers.fr
Claire Maiza
Grenoble INP, Verimag
Grenoble, France
claire.maiza@imag.fr

***Abstract*—We consider the complexity problems of scheduling hard real-time tasks subjected to cache-related preemption delays upon uniprocessor platforms. Several schedulability analysis have been proposed in the literature to explicitly take into account preemption delays due to loss of cache affinity. But, these previous results do not study the complexity of taking scheduling decisions while taking account preemption delays and only focus on classical real-time schedulers (e.g., Rate Monotonic, Earliest Deadline First). In this paper, we focus on the computational complexity of taking scheduling decisions to meet task deadlines while minimizing cache-related preemption delay effects. We design two core cache-related scheduling problems that are the most simple NP-hard problems to cover the most largest set of intractable real-world cache-related scheduling problems. We establish several NP-hardness results in the preemptive and the non-preemptive settings. These results prove that tighter timing analysis leads in practice to harder real-time scheduling problems. These two core NP-hard scheduling problems are the following: (i) scheduling with cache-related preemption delays and (ii) scheduling with information about the cache state and the sequence of requested memory blocks for every task. We also prove for the first problem that neither fixed-task nor fixed-job priority-based scheduling algorithms can be optimal.**

I. Introduction

Real-time systems are subjected to stringent timing constraints due to interactions with complex physical environments. A *schedulability analysis* is performed to validate that all timing constraints will always be met at run-time. Such schedulability analysis requires upper bounds for execution times to be known (WCET - Worst-Case Execution Time). In practice, WCETs are computed by a static analysis of executable program codes, that is called the *timing analysis*. Until recently, real-time system development life cycle copes with these two fundamental analysis in a sequential way: program WCETs are first computed by people handling timing analysis and then subsequently used by people validating systems according to a real-time scheduling algorithm.

Most real-time scheduling-theoretic results assume that all timing penalties due to preemption overheads are integrated into a "magic" WCET for every task [1]. On the one hand, integrating task worst-case timing behaviors within the WCET usually leads to an important overestimation of task execution requirements. As a consequence, the real-time system design must integrate over-dimensioned hardware features. On the other hand, "magic" WCET leads to simpler scheduling problems since tasks are totally independent of each other. As a direct consequence, the most popular scheduling policies do not need any information about task execution requirements for optimally assigning priorities to the tasks. For instance, it is well-known that for uniprocessor platforms, Rate Monotonic (RM) only uses periods to define task priorities, Deadline Monotonic (DM) is based on relative deadlines and Earliest Deadline First (EDF) on absolute deadlines.

Hereafter, we focus on the influence of cache memories in hard real-time scheduling of task systems upon uniprocessor platforms. Cache memories have been introduced in processor chips to reduce the gap between processor and memory speeds. Cache memories, as many other micro-architectural features, increase the processor throughput. Taking explicitly into account cache memories when dealing with real-time scheduling problems offers the opportunity to exploit tighter timing analysis and as a consequence to reduce over-estimations in the system design. But, cache-related preemption delays introduce a circular dependence between timing analysis and the real-time task scheduler. These two topics are usually treated separately in academic work and in "real-world" system design.

In this paper, we investigate the relationships between the scheduling algorithm and cache-related preemption delay estimations. We explicitly consider cache-related preemption delays as a penalty that must be taken into account in the scheduling model. Several schedulability tests have been designed for that purpose, but to the best of our knowledge, no paper investigates the impact of cache-related preemption delays on the computational complexity of taking scheduling decisions.

The computational complexity theory classifies problems according to the resource (time, space,...) needed to solve problem instances of arbitrary size. Complexity results help designers in directing their effort toward those approaches that have the greatest likelihood of leading to useful algorithms [2]. We shall use classical computational complexity classes: NP-hard problems in the weak and in the strong senses. NP-hard problems are optimization problems that cannot be optimally solved in polynomial time. A problem is NP-hard in the weak-sense if it can be solved in pseudo-polynomial time. Whereas problems that are NP-hard in the strong-sense cannot be solved in pseudo-polynomial time and thus an optimal algorithm needs an exponential amount of time to compute an optimal solution. Defining efficient heuristics for such problems involves the determination of properties of underlying combinatorial structure (e.g., graphs, integers, boolean formulas,...) [3].

Our Methodology. We designed two core cache-related scheduling problems that are the most simple NP-hard problems to cover the largest set of intractable cache-related real-world scheduling problems. That is why these core problems voluntary introduce quite restrictive and unrealistic assumptions in order to define the most simple NP-hard problems. Both problems are related to the minimization of worst-case cache effects for meeting hard real-time task deadlines in hard-real time uniprocessor systems. Assumptions are introduced so that core problems can be trivially reused for proving that a wide range of "real-world" cache-related scheduling problems are intractable. Precisely, core problems are trivially reducible to more general problems since they define particular cases of these general problems. Our two core problems and the presented negative results are the following:

- Scheduling with Cache-Related Preemption Delays (CRPD-aware scheduling): every preempted task incurs a CRPD when it is resumed due to a loss of cache affinity. For this problem, we show that neither fixed-job nor fixed-task priority schemes can be optimal. Then, we prove that this problem is NP-hard in the strong sense for the preemptive case.
- Scheduling with cache state information (cache-aware scheduling): the scheduler knows beforehand the sequence of requested memory blocks for every task (computed during the timing analysis) and schedule the tasks to optimize the cache utilization (i.e., minimize the number of cache misses). We show in the preemptive case that the problem is NP-hard in the strong sense. In the non-preemptive case, we show that the problem is NP-hard in the weak sense.

Organization. Section 2 deals with background information and state-of-the-art. Section 3 presents our contributions on the computational complexity of scheduling with CRPD constraints. Section 4 establishes the complexity of scheduling with cache state information both for preemptive and non preemptive task systems. Lastly, Section 5 concludes the paper and indicates some future work.

II. Background

A. Cache memory

Caches are used to bridge the gap between the processor speed and the main memory access time. Caches considerably improve performances but often at the expense of reduced predictability. In order to reduce traffic, the main memory is logically partitioned into a set of memory blocks of equal size. We consider that each logical block is cached as a whole in a cache line.

When the processor requires a memory block, two situations can happen. In the first case, if the block is in the cache: it is a *hit*, no access to the main memory is required and the memory block is sent back to the processor within a short delay. In the other case, it is a *miss*: the corresponding block must be reloaded from the main memory within a large delay. Hennessy and Patterson [4] describe typical values for various caches: a Hit needs between 1 and 4 clock cycles (normally 1), whereas a Miss penalty can be between 8 and 32 clock cycles. Note that, in this paper, we assume that timing analysis are not subjected to timing anomalies [5]. This means that for every task requesting a memory block, a cache hit always leads to a shorter execution time bound than a cache miss.

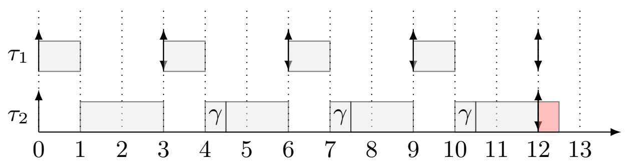
Cache replacement policies have been designed to optimize cache utilization for a single input of block requests. Such an assumption is realistic in most computer programming paradigms. However, it is not anymore realistic for real-time tasks since they are: (i) periodically released, and (ii) scheduled according to a scheduling algorithm that is designed to meet deadline constraints without any timing penalty due to cache misses. Tasks are subjected to numerous preemptions and no single sequence of block requests can be precisely known even in the offline setting (e.g., static cache analysis [6]).

In order to refine the timing analysis, cache-related preemption delays (CRPD) on the execution time due to multitasking may be estimated separately. In order to define tighter WCET analyzer, a classical approach consists in computing the WCET of a given task without any preemption and then computing separately the preemption delays that the tasks may suffer (i.e., CRPD). Cache-Related Preemption Delays are in practice significant and cannot be neglected [7]. Bounding cache-related preemption delays is a classical component of timing analysis [6], [8].

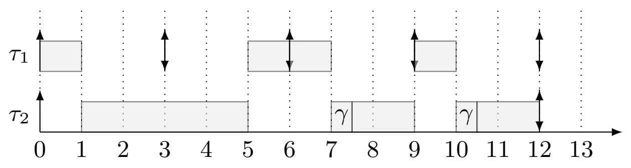
B. State-of-the-art

This section presents known results when dealing with cache-related preemption costs in real-time scheduling problems. We classify these results as follows:

- *Preemption-aware scheduling*. The starting motivation is that arbitrary preemptions can introduce a significant runtime overhead and may cause high fluctuations in task execution times, thus degrading system predictability [9]. Precisely, scheduling policies aim to control the number of preemptions that the tasks may suffer. For instance, schedulability improvement can be achieved by limiting or deferring preemptions [10], [11], [12], [9]. However in all these approaches, scheduling decisions are independent from the WCET, CRPD or any other time cache-related parameters. These approaches define a tradeoff between the fully preemptive and the non preemptive scheduling model.
- CRPD-*aware schedulability*. Analysis and tools are known to compute tight CRPD upper bounds for various



(a) EDF schedule of $\tau_1(1,3,3)$ and $\tau_1(7,12,12)$



(b) Feasible schedule of $\tau_1(1,3,3)$ and $\tau_1(7,12,12)$

Figure 1: Non optimality of task-level fixed and job-level fixed priority rules for scheduling problems with CRPD

cache systems (i.e., direct-mapped, set-associative, full associative, and various replacement policies) [13]. From the schedulability analysis point of view, the WCET and CRPD computed by a timing analyzer are combined to define schedulability tests used to validate real-time systems [14], [15], [13]. Another approach has been proposed by selecting off-line the best possible preemption points in the program codes among a set of potential preemption points (the worst-case context-switch overhead is *a priori* known) [16]. The common point of these approaches is that the scheduler takes its scheduling decisions without any consideration for CRPD timing requirements (i.e., they are not CRPD-aware scheduler). But, CRPD are used at design stage in schedulability tests.

- *cache-aware scheduling*. Only few papers deal with influencing scheduling decisions by taking into account cache state metrics. As indicated in [17], determining how and when to promote jobs to improve cache performance is not straightforward. These approaches are usually based on analytical cache model and on-line monitored counters such as cache miss-rate and concepts such as stack distance profiles or cache footprints (i.e., working set size) [17]. These approaches have been mainly studied in the context of soft real-time systems.

Preemption-aware scheduling will not be covered hereafter since cache related preemption delays are not explicitly taken into account in the scheduling model. In the remainder of this paper, we provide computational complexity results to show that CRPD-aware and cache-aware scheduling problems are intractable (i.e., NP-hard).

III. CRPD-Aware Scheduling

In this section, we focus on the complexity of taking scheduling decisions with CRPD information. We first clearly state the scheduling problem and then prove it to be NP-hard in the strong sense.

A. Simplified CRPD and Task models

We investigate the simplest CRPD model in which the worst-case CRPD is taken into account every time a preemption occurs (for instance, the entire cache must be refilled at each preemption, as in [18]). Hence, the execution requirement C_i corresponds to the WCET when the task is executed alone without preemption and a cache-related preemption delay γ is paid by a task $\tau_i, 1 \leq i \leq n$, every time it is resumed after a preemption. The same penalty γ is taken into account for every task at each preemption point (it may correspond, for instance, to the worst-case preemption delay). **As a consequence, the NP-hardness result presented in this section is also valid for any problem generalization integrating a precise CRPD bound that may be different at each preemption point** (e.g., scheduling models considered in [14], [19], [20], [21], [15], [22], [13]).

We now formally state our first core scheduling problem and then prove its intractability.

Definition 1: Scheduling with CRPD:

- INSTANCE: Finite set of n tasks $\tau_i(C_i, D_i, T_i)$, $1 \leq i \leq n$, with execution requirement C_i (WCET without preemption cost estimated when τ_i is executed fully non preemptively), a relative deadline D_i, a period T_i between two successive releases and a positive number γ representing the worst-case Cache-Related Preemption Delay incurred by τ_i at every resume point after a preemption.
- QUESTION: Is there a uniprocessor preemptive schedule meeting the deadlines?

Notice that this scheduling problem is not restricted to CRPD scheduling problems. It is also applicable to scheduling problems with context switch delays while preempting a job. Such problems have been studied for instance in [23], [10] for uniprocessor real-time systems, without any cache concerns. The next result states that no fixed-job priority rule can be optimal for scheduling tasks with CRPD (or equivalently with context switch delays).

Theorem 1: Neither task-level fixed nor job-level priority rules can be optimal for scheduling tasks with CRPD.

Proof: We provide a simple counter-example of that fact with two tasks: $\tau_1(1,3,3)$ and $\tau_2(8-2\gamma,12,12)$, with an arbitrarily small positive number γ that will represent the CRPD. Let us consider the EDF schedule presented in Figure 1a (Without loss of generality, γ is set to 0.5 in Figure (1a)). In the schedule of τ_2, rectangles labelled by γ represent block reload time delays due to loss of cache affinity that are paid by the preempted task (i.e., cache misses). In practice block reload times are paid when these blocks are referenced. To simplify the graphical presentation in Figure 1, all these block reload times associated to the CRPD are grouped together and depicted just at the resume point. The EDF schedule in Figure (1a) completes τ_2 at time $4 \times C_1 + C_2 + 3\gamma = 12 + \gamma$ and τ_2 misses its deadline. The Figure (1b) depicts a feasible schedule in which τ_2 suffers two preemptions by τ_1 rather than three in the EDF schedule (one γ less to account for). A necessary condition to define a feasible schedule is: τ_2 suffers at most two preemptions. This can be achieved as follows: τ_2 has a higher priority than τ_1 in the interval $[3, 5)$ and τ_1 has a higher priority than τ_2 interval $[5, 6)$ (otherwise, τ_1 will miss its second deadline). In this feasible schedule (1b), τ_2 completes by time $3C_1 + C_2 + \gamma = 12$; the forth job of τ_1 completes by time 10, and thus meets its deadline at time 12. Notice that in Figure (1b), the second preemption of τ_2 can be avoided by raising the priority of τ_2 in the interval $[9, 10.5)$ and thus, ties cannot be broken arbitrarily using a deadline-based scheduling rule. Clearly, fixed-job or fixed-task priority schemes cannot define a feasible schedule in this case. ■

B. NP-*hardness result*

The hardness proof we apply is based on a transformation from the well-known 3-Partition problem, that is recalled hereafter. This problem is strongly NP-Complete meaning that it cannot be solved in polynomial time or in pseudo-polynomial time. Unless P=NP, the number of elementary operations required to solve the 3-partition problem is exponential in the size of the input instance [2]. Hence, assuming that such elementary operations are performed with a fixed amount of time, then the algorithm runs necessarily in exponential-time. We next recall the definition of this classical decision problem; then, we reduce it to our real-time scheduling problem meaning that solving our problem is as hard as solving the 3-Partition decision problem.

Definition 2: The 3-Partition decision problem (i.e., problem [SP15] in [2]):

- Instance: a set A of $3m$ elements, a bound $B \in N$, and a size $s_j \in N$ for each $j = 1..3m$ such that $B/4 < s_j < B/2$ and $\sum_{j=1..3m} s_j = mB$.
- Question: Can A be partitioned into m disjoint sets $A_1, A_2, ..., A_m$ such that, for $1 \leq i \leq m$, $\sum_{j \in A_i} s_j = B$ (each A_i must therefore contain exactly three elements from A)?

Theorem 2: Scheduling with CRPD is NP-hard in the strong sense.

Proof: We transform from 3-Partition as follows:

- $3m$ tasks $\tau_1, \ldots, \tau_{3m}$ with the parameters: $C_i = s_i, D_i = T_i = m(B + 1)$, $1 \leq i \leq 3m$.
- Task τ_{3m+1} with: $C_{3m+1} = D_{3m+1} = 1$ and $T_i = (B + 1)$

We now prove that there is a 3-Partition if, and only if, there is a feasible schedule. By construction, the task set utilization factor without any preemption penalty is exactly 1. Hence, every preemption with a positive CRPD (or equivalently a nonzero context switch delay) will necessarily lead to a deadline failure.

(if part) Assume we have a 3-Partition $A_1, \ldots, A_m$, then schedule τ_{3m+1} as early as possible. The corresponding schedule pattern is presented in Figure 2. All interval between every τ_{3m+1} job's execution (i.e., $[(k-1)(B+1)+1, k(B+1))$, $k = 1..m$) is of length B. Then, schedule any task corresponding to A_k in interval $[(k-1)(B+1)+1, k(B+1))$, $1 \leq k \leq m$. Since there is a 3-Partition, $\sum_{j \in A_k} s_j = B$, for all $k = 1 \ldots m$, thus the corresponding jobs can be scheduled in the interval k without any preemption. All jobs scheduled in these intervals meet their deadlines at time $m(B + 1)$ (i.e., at the end of the last interval). Hence, no cache-related preemption delay will be incurred and all deadlines are met.

(only if part) Assume we have a feasible schedule. Observe that in any feasible schedule, the total workload in interval $[0, m(B + 1))$ is exactly equal to $m \times C_{3m+1} + \sum_{i=1}^{3m} C_i = m + \sum_{i=1}^{3m} s_i = m + mB = m(B + 1)$. Hence, there is no preemption in any feasible schedule since otherwise at least one cache-related preemption delay would be incurred by a task and at least one deadline should be missed (e.g., see the pattern of any feasible schedule presented in Figure 2). Furthermore, due to the 3-Partition problem, execution requirements verify $B/4 < C_i < B/2, 1 \leq i \leq 3m$. Hence, exactly 3 tasks are executed in every interval. Hence, we define a 3-Partition with A_k by selecting the tasks executed in the intervals $[(k - 1)(B + 1) + 1, k(B + 1)), 1 \leq k \leq m$. ■

As a consequence of the previous hardness result, it can be easily proved that there is no *universal* scheduling algorithm taking into account cache-related preemption delays, unless P=NP. We recall that a scheduling algorithm is said to be universal if the algorithm schedules every schedulable task system [24].

Theorem 3: If there exists a universal scheduling algorithm with Cache-Related Preemption Delay then P = NP.

Proof: To prove this theorem, we use a classical proof approach, such as the one presented in [24]. Precisely, we show that if such an algorithm exists, and if it takes a polynomial amount of time (in the length of the input) to choose the next processed job, then P = NP, because we can find a pseudo-polynomial time algorithm to solve the 3-Partition problem. We assume that there exists a scheduling algorithm for the scheduling with *CRPD* problem. We denote this algorithm A. Given an instance of the 3-Partition problem, we define a set I of tasks using the same reduction technique as in the proof of Theorem 2. The tasks are synchronously released. Consequently, to check that every task does not miss its deadline, we only need to study the interval $[0, m(B + 1)]$. Then, we use the scheduling algorithm A to define a schedule and thus we are able to check that all deadlines are met. Since the length of the schedule is $m(B + 1)$ and A is assumed to be a polynomial time algorithm, the whole algorithm for checking deadline is at most pseudo-polynomial (i.e., it is clearly performed in time proportional to mB). Using the reduction techniques proposed in the proof of Theorem 2, the instance I is schedulable by the algorithm A if, and only if, there exists a partition of tasks $\tau_1, \ldots, \tau_{3m}$ into m disjoint sets $A_1, A_2, \ldots, A_m$. Consequently, for each set A_i ($i \in \{1, \ldots, m\}$), we have $\sum_{\tau_j \in A_i} C_j = B$. Thus, the solution delivered by the algorithm A gives a solution to solve the 3-Partition problem. To find this solution, we transform from the 3-Partition problem by simply constructing the set of tasks as in the proof of Theorem 2 and then presenting this task system

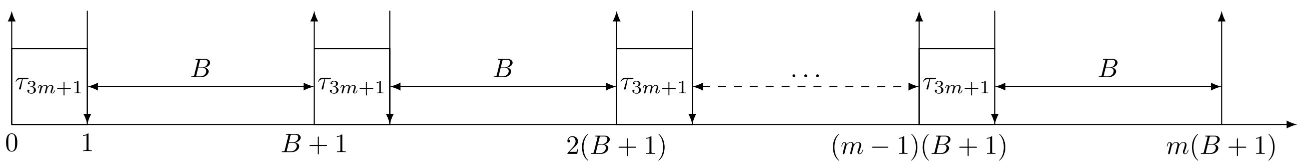


Figure 2: Pattern of feasible schedules in proof of Theorem 2

to the decision procedure based on Algorithm A.
Therefore, we found a pseudo-polynomial time algorithm to solve the 3-Partition problem. However, 3-Partition problem is NP-complete in the strong sense. As a consequence, if the algorithm A exists then P = NP. This is a contradiction and we can conclude that such an algorithm does not exist. ■

Next, we show that the scheduling problem is still hard even if the tasks do not have periodic releases and if preemption delays are equal to one processor cycle (i.e., one unit of time).

Theorem 4: Scheduling with CRPD is NP-hard in the weak sense even if there is two distinct release dates and deadlines, and preemption delay is one unit of time.

Proof: The Partition problem is NP-Complete in the weak sense [2]: given m positive integers $s_1, \dots, s_m$ with $\sum_{i=1^m} s_i = 2B$, decide if there exists a partition of $I = \{1, \dots, m\}$ into two disjoint subsets I_1 and I_2 such that $\sum_{k \in I_j} s_k = B, 1 \leq j \leq 2$. Given an instance of Partition, we define the following scheduling instance with $m+1$ jobs:

- m jobs J_i released at time $r_i = 0$, a deadline equal to $d_i = 2B+1$ and processing times $C_i = s_i, 1 \leq i \leq m$;
- the job J_{m+1} released at time $r_{m+1} = B$ with a deadline $d_{m+1} = B+1$ and $C_{m+1} = 1$.

The previous scheduling instance has two distinct release dates and two distinct deadlines. Preemption delays are equal to one unit of time.

(If part) Assume that we have a Partition (I_1, I_2), then a feasible schedule is obtained by scheduling non preemptively jobs of I_1 in the interval $[0, B)$ and jobs of I_2 in the interval $[B+1, 2B+1)$. Both interval has length B and since $\sum_{k \in I_j} s_k = B, 1 \leq j \leq 2$, then all jobs meet their deadlines.

(Only if part) Assume that we have a feasible schedule. In every feasible schedule:

- job J_{m+1} is scheduled in the interval $[B, B+1)$
- jobs are scheduled non preemptively without any idle time since if one preemption is payed, then one job necessarily misses its deadline due to the preemption delay.

We define a feasible partition by selecting in I_1 the jobs scheduled in $[0, B)$ and in I_2 the jobs scheduled in $[B+1, 2B+1)$. Both interval of length equal to be and the Partition constraint is satisfied: $\sum_{k \in I_j} s_k = B, 1 \leq j \leq 2$. ■

As stated in the previous theorem, the case of non recurring jobs is also hard. Nevertheless, for a finite set of n jobs there are special cases for which EDF creates no preemptions: when all release dates are equal, when all deadlines are equal, when all processing times are equal to one unit of time and when release dates and deadlines are similarly ordered (i.e., $r_i \leq r_j \Rightarrow d_i \leq d_j$, $1 \leq i < j \leq n$). Thus, for all these special cases, EDF is an optimal online scheduling algorithm for the scheduling problem with CRPD.

IV. Cache-Aware Scheduling

In the previous section, the scheduling algorithm only uses a global bound on CRPD and does not exploit neither the sequences of memory blocks requested by the tasks nor the cache states. In this section, we investigate cache-aware scheduling policies, meaning that the scheduler knows the sequence of memory block accesses of each task and then uses this sequence to take scheduling decisions.

Every task code is modeled by a set of requested memory blocks. These blocks may be either instructions or data. The results presented hereafter also hold for separate instruction caches and data caches. We first present assumptions about cache and task models that we use to define a core scheduling problem. The purpose of such a simplistic scheduling model is to simplify as much as possible the proofs without loss of generality. Using such simplified cache and task models, we prove that the corresponding scheduling problems are NP-hard both for preemptive and non-preemptive systems.

A. Simplified Cache and Task Model

Next, we list the simplifications we use to present our computational complexity results. Since, we present a rather simplified scheduling model, **the complexity results can be reused to prove the hardness of more general task and cache systems in which all these assumptions are no longer assumed**.

Assumption 1: The cache memory consists of a single cache line. A hit is performed at no cost and the miss penalty is equal to a constant BRT (Block Reload Time).

By considering a cache memory with only one cache line, Assumption 1 defines the simplest particular case that covers all cache types: direct-mapped, set-associative and fully-associative caches. Precisely, it is always possible to define input instances (i.e., cache memory accesses and mapping of blocks in the main memory) that lead a set-associative cache as well as a fully-associative cache to reload/evict the same blocks as for a direct-mapped cache. Obviously, such an assumption only focuses on corner cases that lead to worst-case cache utilization. Hence, the presented NP-hardness results is valid for all cache strategies (i.e., direct-mapped, fully-associative, and set-associative). Furthermore, this assumption also exhibits that the computational complexity of scheduling under cache constraints is independent of the cache size.

We also assume a basic task model in which every job has a single execution path in its control flow graph. This is once again the most simple task structure that can be considered. If references of memory blocks are represented by letters in a finite alphabet Σ, then the sequence S_i of requested memory blocks in every task τ_i can be modeled by a word of Σ^*.

Assumption 2: The Control Flow Graph of every job consists of a single execution path with loop unrolling.

We only consider a finite set of jobs with individual deadlines. We only know for every job: its execution requirements (i.e., WCET) assuming cache hit for every requested memory block and the sequence of used memory blocks. Thus, the instant at which a memory block is requested is not defined in the task model. Finally, we make a third assumption to limit preemptions just before specific points in the code.

Assumption 3: In the preemptive setting, job preemptions can only occur just before requesting the next memory block.

Limiting preemption points in the code can only improve the number of cache hits since the cache content can only change at discrete points in time [9]. Next, we consider only non recurring jobs subjected to individual deadlines. This choice is only made to define the most simplified task model for which cache-related scheduling problems are still NP-hard.

Definition 3: A job J_i is defined by $J_i(C_i, S_i, D_i)$, where:

- C_i is the WCET assuming a cache hit for any reference.
- S_i is a string from Σ^*, denoting the sequence of memory blocks used during the job execution.
- D_i is the relative deadline of the job.

Blocks referenced in S_i of task τ_i may be instruction or data blocks. For instruction cache, the intersection $S_i \cap S_j$ represents the shared code (e.g., shared functions, library code, operating system) and for data cache it represents common data. For data cache, sequence S_i can be totally arbitrary, without any pattern constraints. Whereas for instruction cache, blocks that are referenced several times in sequences $S_i, 1 \leq i \leq n$ must necessarily correspond to: function call or code within loops inside a task. Without modifying Definition 3, the sequence S_i can represent requested blocks corresponding simultaneously to instructions and data.

B. Preemptive scheduling

We can now state the simplified scheduling problem that will be proved to be NP-hard.

Definition 4: The scheduling problem with cache memory ($SDCM$) is:

- INSTANCE: a finite alphabet Σ, a finite set of n jobs $J_i(C_i, D, S_i)$ released at time 0, with execution requirement C_i and sequence of used memory blocks $S_i \in \Sigma^*$, a common deadline D and a positive number BRT.
- QUESTION: Is there a uniprocessor preemptive schedule meeting the overall deadline D so that every hit in the cache is performed without any penalty and every miss has a penalty of BRT units of time?

Note that C_i is defined by the length of S_i (i.e., $C_i = |S_i|$, by Assumption 2). Hence, this WCET assumes that all requested memory blocks are hits in the cache.

Consider the following instance in which all blocks are shared by all tasks (e.g., data cache): $\Sigma = \{a, b, c\}$, 3 jobs with a block reload time $BRT = 0.5$ and a common deadline $D = 21$:

- $J_1(5, 21, babcc)$
- $J_2(6, 21, aaccbc)$
- $J_3(5, 21, bacca)$

Since, $\sum_i C_i = 16$ and there is exactly 16 memory block requests, then any feasible schedule must experience at most 10 cache misses to meet the overall deadline D. Without loss of generality, we assume in this example that every portion of code requests one memory block for 1 unit of time. If the corresponding block is in the cache there is no additional timing penalty, but if it is a miss, the operation requires 0.5 unit more due to the main memory access (i.e., BRT). These values have been chosen for the ease of graphical representation in Figure 3. In Figure 3, the first two blocks that are requested by J_1 and J_2 are not cached. As a consequence, their executions incur a penalty of BRT units of time; but, when J_3 first requires Block b, it results in a cache hit, and thus no penalty is incurred. Clearly, it is easy to define a schedule in which there are only cache misses with a total length of 24 (i.e., BRT has been incurred at every cache access). As a consequence, at least one job will miss the deadline.

Theorem 5: Task-level and job-level fixed priority schedulers are not optimal for the scheduling of tasks with cache memory (i.e., $SDCM$ problem).

Proof: We provide a simple counter-example for job-level fixed priority schedulers. Consider a memory cache with $BRT = r$, where r is an arbitrary positive number; two jobs synchronously released and having the same relative deadline equal to $D = 2C + 4r$: $J_1(C, D, aba)$ and $J_2(C, D, bab)$, where C is an arbitrary positive number. A fixed-task priority scheduler as well as a job-level fixed priority scheduler will define a priority ordering before starting one of these two jobs. Without loss of generality, assume that J_1 obtains a higher priority than J_2. The sequence of memory blocks in the cache will be: $\sigma = ababab$ leading to 6 cache misses and no hit. As a consequence, the schedule has a length of $2C + 6r$. If $C = 3$ and $r = 0.5$, the length of the schedule will be 9 units of time. The shortest schedule of J_1 and J_2 is obtained using a full dynamic priority scheduler leading to the sequence depicted in Figure 4, assuming once again $C = 3$ and $r = 0.5$. Clearly, J_1 has a higher priority than J_2 in the interval $[0, 3]$, then at time 3, the priority of J_2 is raised to a higher priority than the one of J_1, and finally reduced to a lower priority at time 5.5. There are two cache hits and four cache misses leading to a schedule of length $2C + 4r$. This is the smallest number of hits that can be achieved while scheduling these two jobs. This counter example is also valid for fixed-task priority schedulers since they are a particular case of job-level fixed-priority schedulers.

■

To prove that the $SDCM$ problem is strongly NP-hard, we will make a reduction from Shortest Common Supersequence, denoted SCS hereafter, that is known to be NP-hard in the strong sense [25], [2]. We first recall basic definitions before the problem statement.

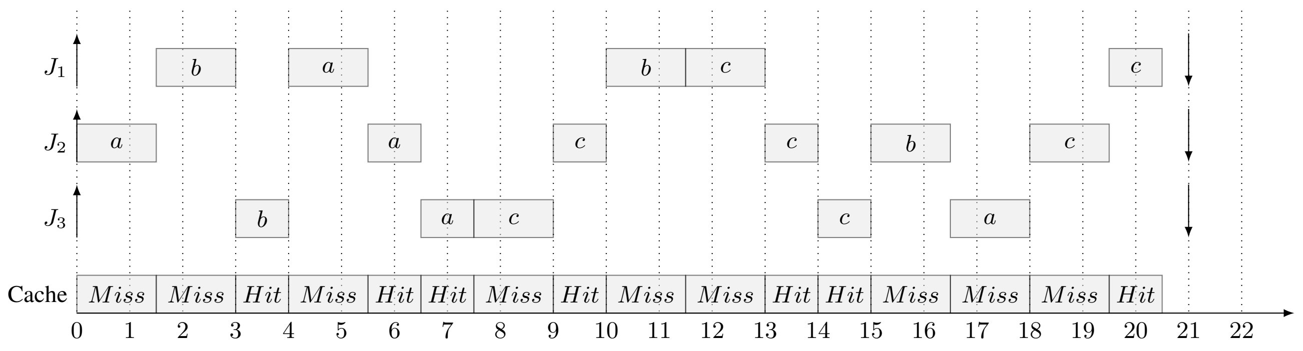


Figure 3: Schedule of 3 jobs using cache blocks a, b and c ($C = 1, r = 0.5$). A timing penalty is incurred at each cache miss.

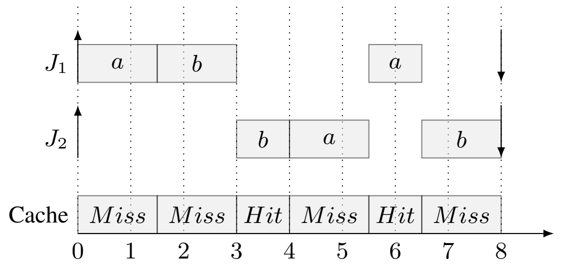


Figure 4: Task-level fixed and job-level fixed priority schedulers are not optimal ($C = 1, r = 0.5$)

Algorithm 1: SCHED(J,w)

```
input :
        $J_j(C_j, S_j), 1 \leq j \leq n$ ;
        $w$ : Shortest Common Supersequence of $s_i$'s;
$k_j := 1 \quad \forall j = 1..n$ ;
foreach $i := 1, \ldots, |w|$ do
    /* For each block in the Shortest Common Supersequence $w$ */
    foreach $j := 1, \ldots, n$ do
        /* For each job in $J$ */
        while $s_{j,k_j} = w_i$ do
            /* the job uses block $w_i$ until next preemption point */
            DISPATCH($J_j$); /* Execute $J_j$ up to its next preemption point in time */
            $k_j := k_j + 1$;
        end
    end
end
```

Given a finite sequence $\sigma = s_1, s_2 ..., s_m$, we define a subsequence σ' of σ to be any sequence which consists of σ with between 0 and m terms deleted. We write $\sigma' < \sigma$. Given a set $R = \{\sigma_1, \ldots, \sigma_p\}$ of sequences, a Shortest Common Supersequence of R, denoted $SCS(R)$, is a shortest sequence such that every σ_i, $1 \leq i \leq p$ is a subsequence of $SCS(R)$ (i.e., $SCS(R) > \sigma_i, 1 \leq i \leq p$). For instance $SCS(\{abbb, bab, bba\}) = abbab$.

Definition 5: The Shortest Common Supersequence (SCS) is (i.e., problem [SR8] in [2]):

- INSTANCE: a finite alphabet Σ, a finite set R of strings from Σ^*, and a positive integer K.
- QUESTION: Is there a string $w \in \Sigma^*$ with $|w| \leq K$ such that each string $x \in R$ is a subsequence of w, i.e, $w = w_0 x_1 w_1 x_2 \ldots x_k w_k$ where each $w_i \in \Sigma^*$ and $x = x_1 x_2 \ldots x_k$?

The SCS problem is NP-Complete in the strong sense [25] meaning that there does not exist a polynomial time algorithm to solve it. Polynomial special cases are known if $|R| = 2$ or if all $x \in R$ have $|x| \leq 2$. Furthermore, it is also MAX-

SNP hard [26], meaning that it is hard to approximate (i.e., there does not exist a polynomial time approximation scheme - PTAS), unless $\mathsf{P} = \mathsf{NP}$.

Theorem 6: The SDCM scheduling problem is NP-hard in the strong sense.

Proof: We reduce the SCS problem to the $SDCM$ scheduling problem. We define an instance of the $SDCM$ problem from an arbitrary instance of SCS as follows:

- The finite alphabet Σ is used to defined memory blocks of the considered cache line.
- To every $x \in R$ we define a job J_i with an execution requirement $C_i = |x|$ and a cache request sequence $S_i = x$ (i.e., $J(|x|, D, x)$, where D is the common deadline.
- The deadline $D = \sum_{x \in R} |x| + K.BRT$, where BRT is assumed to be a positive number.

We now prove that there exists a solution to the SCS instance, if and only if, there exists a solution to the $SDCM$ instance. The principle of the transformation is to establish that the shortest common supersequence corresponds to the schedule with the minimum number of memory accesses (i.e., cache misses).

(if part) There exists a shortest common supersequence w of length K or less. By construction, w is a shortest common supersequence of jobs S_i, $1 \leq i \leq n$. We use w to schedule jobs so that cache accesses exactly follow w. We describe the simple scheduling algorithm. Let $s_{j,k}$ be the k^{th} block requested in S_j for Job J_j and let k_j be the next requested block in that sequence: starting from w_1, we schedule every job so that $s_{j,k_j} = w_1$ for one unit of time in arbitrary order. For these scheduled jobs, we increment indexes k_j, $1 \leq j \leq n$. Then, the same scheduling rule is applied to every subsequent w_j, $j \leq |w|$. Algorithm 1 presents the corresponding pseudo-code. In this algorithm, at most one cold cache miss is paid for every w_i (i.e., exactly one if $w_i \neq w_{i-1}$, zero otherwise since the block is already cached). The number of loaded blocks in the cache line is thus bounded by K, the length of the supersequence. As a consequence, the length of this schedule is bounded by $\sum_{i=1}^{n} C_i + K.BRT \leq D$ and the common deadline is met for all jobs.

(Only if part) Assume that we have a schedule meeting the overall deadline D. Let σ be the corresponding sequence of block requests in the considered cache line. Necessarily, $\sum_{i=1}^{n} C_i + K.BRT \leq D$, where K is defined by construction.

Without loss of generality, we assume that subsequences in Σ composed of one single memory block (i.e., letters) are sorted using a wrapping around technique on job indexes. Such an ordering does not change the number of cache hits or misses since reordered blocks are consecutive and identical in σ. Notice that by construction σ contains all $x \in R$ since $|\sigma| = \sum_{x \in R} |x|$.

We start from the beginning of $w_0 = \sigma$ by repeating the following step: at step i we aggregate all subsequent identical blocks corresponding to different jobs for defining w_i (i.e., only one block of a given job will be aggregate at every step). We denote w, the sequence obtained from this process. Due to the assumption on the ordering of identical blocks, aggregated letters are always consecutive in the sequence.

Steps	Sequences
0	$w_0 = abbaaaccbcccbacc$
1	$w_1 = abbaaaccbcccbacc$
2	$w_2 = abaaaccbcccbacc$
3	$w_3 = abaccbcccbacc$
4	$w_4 = abacbcccbacc$
5	$w_5 = abacbcccbacc$
6	$w_6 = abacbcbacc$
7	$w_7 = abacbcbacc$
8	$w_8 = abacbcbacc$
9	$w_9 = abacbcbac$

Table I: Construction of a common supersequence (w_9) starting from the cache request sequence (w_0) associated to an arbitrary schedule

The two next claims prove that w is a supersequence of at most length K.

Claim 1: w is a supersequence. (By Induction) Initially, all $x \in R$ are subsequences of w_0. Let $w_i, i \geq 0$ be the sequence at a step verifying the induction hypothesis. Consider the obtained sequence w_{i+1} computed from w_i: since aggregated letters come from different jobs by construction, then it follows that all $x \in R$ are subsequences of the sequence w_{i+1}. Thus, the sequence w obtained by this process is a supersequence of all $x \in R$.

Claim 2: the length of w is at most K. The worst-case number of cache misses is necessarily obtained when all subsequent letters in w are distinct. Without loss of generality, we consider hereafter this worst-case scenario. As a consequence, $|w|$ corresponds to the number of cache misses which is less than or equal to K by construction, since the overall deadline $D = \sum_{x \in R} |x| + K.BRT$ is met. Hence, w is a common sequence of length at most K, in the considered worst-case scenario. ■

We illustrate the previous reduction by considering the schedule presented in Figure 3 (i.e., $\Sigma = \{a, b, c\}$, 3 jobs: $J_1(5, babcc)$, $J_2(6, aaccbc)$, $J_3(5, bacca)$). The construction of a common supersequence from a feasible schedule is illustrated in Table I. At each step, subsequent identical blocks coming from different jobs are aggregated.

C. Non Preemptive Scheduling

Single processor non preemptive scheduling (without cache delays) is a well studied problem. Non preemptive scheduling of a finite set of jobs with release dates and deadlines is already known NP-hard (see also [2], problem [SS1]). If the jobs are a priori known and (i) they are simultaneously available and (ii) subjected to individual deadlines, then EDF (also known as Jackson's rule) is an universal scheduling algorithm.

We assume a simplified task model in which every task accesses only one memory block. We still continue to assume, as in the preemptive case, that the cache memory consists in a single cache line containing one memory block.

Definition 6: The non preemptive scheduling problem with cache memory ($NPSCM$) is:

- INSTANCE: a finite alphabet Σ, a finite set of n jobs $J_i(C_i, D_i, S_i)$, with an execution requirement C_i, a

deadline D_i, the used memory blocks $S_i \in \Sigma$, and a positive number BRT (Block Reload Time).
- QUESTION: Is there a uniprocessor non preemptive schedule meeting deadlines D_i for every job so that every hit in the cache is performed without any penalty and every miss has a penalty of BRT units of times?

The hardness proof is based on a transformation from the following sequencing problem that is known to be NP-hard in the weak sense [27], [2]:

Definition 7: Sequencing with deadlines and setup times is defined as follows (i.e., problem $[SS6]$ in [2]):

- INSTANCE: a set C of "compilers", a set T of tasks, for each $t \in T$ a length $l(t) \in Z^+$, a deadline $d(t) \in Z^+$, and a compiler $k(t) \in C$ and for each $c \in C$ a setup time $l(c) \in Z^+$.
- QUESTION: Is there a one processor schedule σ for T that meets all task deadlines and that satisfies the additional constraint that, whenever two tasks t and t' with $\sigma(t) < \sigma(t')$ are scheduled "consecutively" (i.e., no other task t'' has $\sigma(t) < \sigma(t'') < \sigma(t')$) and has different compilers (i.e., $k(t) \leq k(t')$), then $\sigma(t') \geq \sigma(t) + l(t) + l(k(t'))$?

The next result states that the NPSCM scheduling problem is NP-hard in the weak sense.

Theorem 7: The non preemptive scheduling problem with cache memory (NPSCM) is NP-hard in the weak sense.

Proof: We transform from the problem $[SS6]$ that is NP-Complete in the weak sense, even if setup times are identical [2]. Hereafter, we consider a constant setup time L for all $c \in C$. Let us consider an arbitrary instance of $[SS6]$ to define an instance of our scheduling problem:

- $\Sigma = C$
- For every task $t \in T$, we define a job J_i with parameters $C_i = l(t)$, $D_i = d(t)$ and $S_i = k(t)$. Thus, we assume that every job uses only one memory block mapped into the cache line.
- $BRT = L$ is the Block Reload Time.

Clearly, a task compiler corresponds to the memory block used by a job to be cached in the cache line. The setup time corresponds to the block reload time whenever a memory block is not in the cache line. The block reload time in our scheduling problem with cache memory corresponds exactly to the setup time in the sequencing problem. As a consequence, both problems are equivalent. Hence, the problem $[SS6]$ has a solution if, and only if, the corresponding scheduling problem has a feasible solution. ■

The previous transformation only establishes that the non-preemptive cache-aware scheduling problem is NP-hard in the weak sense. The existence of a pseudo-polynomial time algorithm cannot be excluded. Nevertheless, we think that as in the preemptive case, this problem is harder but we currently have no formal proof that it is NP-hard in the strong sense.

V. Conclusion

In this paper, we define two core cache-related scheduling problems: (i) scheduling with cache-related preemption delays and (ii) scheduling with cache information. We establish several negative computational complexity results both for preemptive and non-preemptive scheduling problems. We also show that popular fixed-task and fixed-job priority rules cannot be used to define an optimal CRPD-aware scheduler. As a consequence, tighter timing analysis leads to harder scheduling problems. In other words, taking explicitly into account cache memories in scheduling problems cannot be achieved using straightforward generalizations of well-known uniprocessor scheduling-theoretic results.

Per se, performances of real time systems simultaneously depend on WCET (optimized compiler, timing analyzer), cache memory management (replacement policy, locking and partitioning techniques), the schedulability analysis used to validate the system, and last but not least, the scheduler that takes scheduling decisions at run-time. Overestimations are introduced by all these techniques in order to design predictable systems. In order to reduce these underlying overdimensioning effects, we believe that improvement in the design of real-time systems can only be achieved by tackling these problems simultaneously.

Future work are to study which tradeoff must be made in order to efficiently coping with a cache memory in real-time systems. Typical questions are: (i) Which kind of system model is required: fine-grained or coarse scheduling models? (ii) How to jointly handle task memory accesses (e.g., replacement policy, locking and partitioning techniques) and task scheduling (e.g., controlling preemptions according to the cache state and the schedulability issue)?

References

[1] J. Schneider, "Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems," in *In Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'2000)*, 2000, pp. 195–204.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[3] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, ser. The IBM Research Symposia Series, R. Miller, J. Thatcher, and J. Bohlinger, Eds. Springer US, 1972, pp. 85–103.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.

[5] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[6] 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.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[7] S. Basumallick and K. Nilsen, "Cache issues in real-time systems," in *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[8] R. Wilhelm and D. Grund, "Computation takes time, but how much?" *Commun. ACM*, vol. 57, no. 2, pp. 94–103, Feb. 2014.

[9] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Trans. Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[10] J. Lee and K. Shin, “Preempt a job or not in edf scheduling of uniprocessor systems,” *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1197–1206, May 2014.

[11] J. Marinho and S. Petters, “Job phasing aware preemption deferral,” in *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, Oct 2011, pp. 128–135.

[12] J. Marinho, V. Nelis, S. Petters, and I. Puaut, “Preemption delay analysis for floating non-preemptive region scheduling,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, March 2012, pp. 497–502.

[13] S. Altmeyer, R. Davis, and C. Maiza, “Improved cache related preemption delay aware response time analysis for fixed priority preemptive systems,” *Real-Time Systems*, vol. 48, no. 5, pp. 499–526, 2012.

[14] C.-G. Lee, J. Hahn, Y.-M. Seo, S.-L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C.-S. Kim, “Analysis of cache-related preemption delay in fixed-priority preemptive scheduling,” *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700–713, Jun 1998.

[15] Y. Tan and V. Mooney, “Timing analysis for preemptive multitasking real-time systems with caches,” *ACM Transactions Embedded Computing Systems*, vol. 6, no. 1, Feb. 2007.

[16] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, “Optimal selection of preemption points to minimize preemption overhead,” in *23rd Euromicro Conference on Real-Time Systems (ECRTS’11)*, July 2011, pp. 217–227.

[17] J. Calandrino and J. Anderson, “On the design and implementation of a cache-aware multicore real-time scheduler,” in *21st Euromicro Conference on Real-Time Systems (ECRTS’09)*, July 2009, pp. 194–204.

[18] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings, “Adding instruction cache effect to schedulability analysis of preemptive real-time systems,” in *Proceedings IEEE Real-Time Technology and Applications Symposium*, Jun 1996, pp. 204–212.

[19] H. Tomiyama and N. D. Dutt, “Program path analysis to bound cache-related preemption delay in preemptive real-time systems,” in *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, ser. CODES ’00, 2000, pp. 67–71.

[20] H. S. Negi, T. Mitra, and A. Roychoudhury, “Accurate estimation of cache-related preemption delay,” in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS ’03, 2003, pp. 201–206.

[21] J. Staschulat, S. Schliecker, and R. Ernst, “Scheduling analysis of real-time systems with precise modeling of cache related preemption delay,” in *Proceedings. 17th Euromicro Conference on Real-Time Systems(ECRTS 2005)*, July 2005, pp. 41–48.

[22] C. Maiza-Burguière, J. Reineke, and S. Altmeyer, “Cache-related preemption delay computation for set-associative caches—pitfalls and solutions,” in *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

[23] P. Yomsi and Y. Sorel, “Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems,” in *Real-Time Systems, 2007. ECRTS ’07. 19th Euromicro Conference on*, July 2007, pp. 280–290.

[24] K. Jeffay, D. Stanat, and C. Martel, “On non-preemptive scheduling of periodic and sporadic tasks,” *proc. Real-Time Systems Symposium*, pp. 129–139, 1991.

[25] D. Maier, “The complexity of some problems on subsequences and supersequences,” *J. ACM*, vol. 25, no. 2, pp. 322–336, 1978.

[26] T. Jiang and M. Li, “On the approximation of shortest common supersequences and longest common subsequences,” *SIAM Journal on Computing*, vol. 24, no. 5, pp. 1122–1139, 1995.

[27] J. Bruno and P. Downey, “Complexity of task sequencing with deadlines, set-up times and changeover costs,” *SIAM Journal on Computing*, vol. 4, no. 7, pp. 393–404, 1978.