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

Rapport de Recherche

N° 03 - 2015

28/09/2015

Cache-Related Preemption Delays and Real-Time Scheduling: A Survey for Uniprocessor Systems

Guillaume PHAVORIN, Pascal RICHARD

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

The trend in nowadays real-time embedded systems is to use commercial off-the-shelf components, and in particular CPUs with cache memories. But because of the way the cache is working, additional delays known as Cache-Related Preemption Delays (CRPDs) might occur as soon as preemptive scheduling is considered. These CRPDs make the predictability problem more complex and may even threaten the system schedulability. This article presents different existing strategies to deal with CRPD issues. These strategies can focus on reducing the CRPDs at the cache level or can work at the scheduling level to control the number of preemptions. Combinations between those different methods are also presented.

CONTENTS

I Introduction 3

II Basic notions 4
- II-A Caches . . . 4
- II-B Preemption delays . . . 4
- II-C Scheduling . . . 5
- II-D Evaluation techniques . . . 5
- II-E Technique roadmap . . . 6

III Timing Analysis 6
- III-A General WCET computation issue . . . 6
- III-B WCET cache analysis . . . 7
- III-C Including Cache-Related Preemption Delays into the WCET . . . 8

IV Memory management 9
- IV-A Cache partitioning . . . 9
- IV-B Cache locking . . . 10
- IV-C Memory layout . . . 11

V Enhanced task models 12
- V-A Cache-aware scheduling analysis . . . 12
- V-B Limited preemption scheduling . . . 13
- V-C Cache-aware scheduling . . . 14

VI Prospects 14

VII Conclusion 15

Cache-Related Preemption Delays and Real-Time Scheduling: A Survey for Uniprocessor Systems

Guillaume Phavorin, Pascal Richard
LIAS/Université de Poitiers
Poitiers, France
{guillaume.phavorin, pascal.richard}@univ-poitiers.fr

I. Introduction

Nowadays, embedded systems are widely spread. As they are made of more and more real-time applications, increasing processing capacity is needed. For uniprocessors, performances have been increased over the years by speeding up the processor frequency and introducing micro-architectural features such as pipelines and branch prediction. But, as a result, the gap between the processor speed and the main memory access time has increased exponentially. So, cache memories had to be introduced to bridge this gap. As most real-time applications are critical, it must be proved that they meet their timing constraints: schedulability tests are used to verify beforehand the system validity. Moreover, such systems must be predictable: test results must be valid for the worst-case possible behaviours.

Real-time embedded systems (RTES) use more and more commercial off-the-shelf components (COTS) as they allow significant cost savings, see for example the report from the Federal Aviation Administration [1]. Most commercial processors incorporate cache memories to increase performances. But on the other hand, caches make the predictable problem even more complex: instruction and data loading depend on whether the reference is found in the cache or has to be reloaded from the main memory. Furthermore, because of preemptions, some data may have been thrashed from the cache and so additional reloads might be performed from the main memory, leading to extra delays referred to as Cache-Related Preemption Delays (CRPDs). These additional delays can represent as much as 40% of a task worst-case execution time (WCET) [82]. The easiest way to deal with the CRPD matter is to disable the cache. It is however not always possible on modern hardwares, and in any case, it leads to a drop in performances.

Usually, in real-time scheduling, hardware-related costs (including switch context, scheduler costs, CRPD...) are assumed to be part of the WCET of each task. So, from the scheduling point of view, preemptions are performed at no cost. As a consequence, the scheduling problem becomes easier as task behaviours are independent from each other. But on the other hand, this approach often results in overestimated execution times: WCETs are increased and so is the processor utilization. That leads to a waste of resources as the system has to be oversized: task average execution times, because of the cache, will be far smaller than the WCETs, and as a result the CPU will be underutilized. So, an other approach is to dissociate WCETs and CRPD. But as a consequence, task behaviours are no more independent from each other: a circular dependency is introduced between the timing analysis and the schedulability analysis.

To reduce the pessimism introduced by caches, numerous strategies have been proposed. Some consist only in bounding the CRPD and incorporate it either directly in the WCET or rather in the schedulability analysis. Other strategies focus on reducing such sources of pessimism: the cache behaviour can be modified to reduce or even eliminate possible cache thrashing by other tasks, or the scheduling policy can be adapted to reduce the number of preemptions and/or reducing the CRPD. Although, many of these methods are mutually dependant, they have been mostly studied in isolation either from the timing issue or the schedulability issue.

a) Goal.: So, the purpose of this article is twofold. First we present the known strategies dealing with the CRPD problem. Some of them aim at improving cache-related delay estimation to tighten the bounds on both the WCET and the CRPD. Other approaches prefer to focus on scheduling algorithms to avoid costly preemptions whenever it is possible and as a result increase the system schedulability. Then, we focus on possible combination between those techniques and we present the schedulability tests associated with them.

b) Assumptions.: In this paper, for sake of simplicity, no pipeline is considered. Moreover, we deal only with fully timing compositional architectures (see the architecture classification presented in Cullmann *et al.* [33]), i.e. processors for which no timing anomaly occurs (see [66] and [98] for further information on that matter). We also mainly consider instruction caches as they are more commonly found in embedded architectures than data caches. Finally, we focus on fixed-task priority scheduling to illustrate the different approaches presented in this paper.

c) Organization.: This survey is organized in five main parts. First, we introduce basic notions regarding caches and real-time scheduling. Then, we present how task worst-case execution times (WCETs) can be computed when cache memories are involved. In Section IV, we focus on approaches to deal with cache issues directly at the memory level. In

Section V, we handle the problem of scheduling with cache issues. Finally, we briefly present some possible combinations between the different approaches presented before.

II. Basic notions

The preemption cost problem can be addressed from different angles. The focus can be on a tight estimation and possible reduction of the number of preemptions a system may experience in the worst case. On the other hand, a bound on the preemption delay can be computed to be later taken into account in the schedulability analysis. In this paper, we mainly focus on the latter approach.

We introduce here some basic notions regarding Cache-Related Preemption Delays (CRPDs). First, we present very briefly cache memories and how they work. Then, we discuss the problem of preemption delays and particularly CRPDs. In a third part, we provide notions about real-time scheduling and how it can be extended to handle CRPDs. Finally, we summarize the most common techniques used to evaluate the different approaches presented hereafter.

A. Caches

Caches are fast accessible memories, much faster than the main memory but still slower than the registers. They are used to save data loaded from the main memory. Thus, a later access to this data by the CPU will be served directly from the cache (being a cache *hit* opposed to a cache *miss*), resulting in time earning (see [78]) and less power consumption (as stated in [113]). For example, according to [50], a hit needs between 1 and 4 clock cycles to be served, whereas a miss can cost up to 32 cycles. The effectiveness of caches is based on the principle of reference locality: a resource is more likely to be referenced if another resource near to it has been referenced recently (*spatial* locality), e.g. sequential instructions, and already-referenced ressources are more likely to be re-referenced in a short laps of time (*temporal* locality), e.g. in loops.

Caches can be classified as *instruction caches*, *data caches* or *unified caches* depending on the kind of resources they store: respectively program instructions, program data or both of them. In this paper, for sake of simplicity and if not stated otherwise, we only consider one cache level and very often only instruction caches.

A cache is divided into lines of equal size, each of which can store one memory block loaded from the main memory. A memory block is a logical partition of the main memory: it is the smallest amount of bytes which can be loaded at a time from the main memory. It can contain several data (for data caches) or instructions (for instruction caches), to increase spatial locality. As depicted in Figure 1, different strategies can be used to decide to which cache line a given memory block will be mapped:

- *direct-mapped* caches: a memory block has only one possible location into the cache, depending on its address, as shown in 1(a),
- *fully-associative* caches: as depicted in 1(b), a memory block might be mapped to every line, depending on the cache history and a replacement policy,

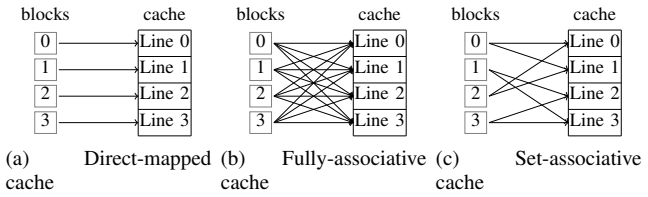


Figure 1. Mapping examples for a Direct-Mapped Cache (left), Fully-Associative Cache (center) and 2-Ways Set-Associative Cache (right)

- *set-associative* caches, which is the intermediate case: the cache is divided into sets of equal number of lines and a given memory block can only be mapped to a particular set, depending on its address, but then be placed anywhere into that particular set, depending on the cache history and a replacement policy, see 1(c).

Details on replacement policies used for set- and fully-associative caches can be found in [50].

Different metrics have been proposed in the literature to characterize how the cache behaviour may affect a task. The simplest one is the task Working Set Size (WSS), introduced in [34]: it corresponds basically to the amount of cache lines accessed by the task during its execution. It is however a raw indicator, representing the average behaviour of a task. More precise but complex metrics can be used instead: the Stack Distance (see [74]), which corresponds to the number of different cache lines accessed between two consecutive accesses to a same reference, or the Reuse Distance (see [14]), which is similar to the Stack Distance with the exception that the constraint of intermediary accesses being mapped to different cache lines is released. These last two metrics are particularly useful when dealing with fully- or set-associative caches.

B. Preemption delays

When a preemption occurs, additional delays have to be considered alongside the normal execution time of the task, due to context switches, extra-bus interference cost... (see [27]). In this paper we only study the *cache-related preemption delay* (CRPD) which corresponds to the time needed to reload cache lines that have been evicted by preempting tasks. Indeed, other costs are less penalizing and can often be bounded by a constant (see [7]).

As stated in [13], the interference the cache has on a task execution time can be:

- *intrinsic* (referred also to as *intra-task*): that corresponds to memory block reloads because of the task structure and the hardware (multiple instructions or data might be mapped to the same cache line). Those costs are usually accounted for directly in the WCET.
- *extrinsic* (referred also to as *inter-task*): that corresponds to the damage due to other tasks that may preempt the considered task. These costs correspond to the CRPD.

Category	Technique			References	Section
memory management	cache partitioning	fully-partitioning		[56, 79, 87, 6]	IV-A
		hybrid-partitioning		[107, 21, 106]	
	cache locking	full locking	static	[31, 41, 64, 53]	IV-B
			dynamic	[10, 88, 89, 53]	
		partial locking	static	[35]	
			dynamic	[36]	
	partitioning + locking			[113]	
	memory layout	code positioning		[111, 76, 65, 40, 77]	IV-C
		task positioning		[45, 8, 69]	
WCET	WCET with cache analysis			[118, 49, 39, 109, 92, 100]	III-B
	WCET with CRPD	preempting task		[13, 110]	III-C
		preempted task		[102, 3, 28]	
		both tasks		[80, 101, 116]	
scheduling	schedulability analysis	preempting task		[25, 24]	V-A
		preempted task		[58]	
		both tasks		[105, 5, 52, 67]	
	preemption control	preemption thresholds with CRPD		[20, 114]	V-B
		floating-Non Preemptive Region with CRPD		[93, 73, 72]	
		Fixed Preemptive Points with CRPD		[103, 4, 17, 16, 83, 32]	
	optimal	cache-aware scheduling		[85, 86]	V-C

scheduling ↓ → timing

Table II. OVERVIEW OF DIFFERENT METHODS TAKING THE CACHE INTO ACCOUNT

C. Scheduling

In this paper, we consider only uniprocessors and mainly hard real-time periodic tasks. The set of tasks assigned to a processor is noted τ. A task $\tau_i(C_i, D_i, T_i)$ belonging to τ is characterized by the following timing constraints:

- its *worst-case execution time* (WCET) C_i, i.e the maximal time needed by the processor to execute the task. Traditionally, this WCET is considered to account for every potential delay the task may experience. However, when the WCET does not account for CRPDs, as in Subsection V-A, it will be denoted $\hat{C}_i$.
- its *period* T_i, i.e. the delay between the releases of two consecutive jobs of the task,
- its relative *deadline* D_i, which is the time, following the task arrival time (release), at which the task should be completed. A deadline is said to be *implicit* if it is equal to the task period, *constrained* when it is smaller than the task period, or *arbitrary* when it can exceed the task period.

As explained in [27], a *scheduling algorithm* is used to decide, at each instant, which task has to be executed in order to respect timing constraints. Moreover, we focus mostly on online scheduling and in particular on *fixed-task* priority scheduling (FTP), i.e. all jobs of a task have the same priority which does not change throughout the application life.

For a given taskset, a *schedulability analysis* can be conducted to determine if a given scheduling algorithm does not violate any timing constraint for this system. For FTP scheduling and tasks with constrained deadlines, we will mostly use the response time analysis (RTA) introduced in [54]:

$$\forall i, R_i \leq D_i \tag{1}$$

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{2}$$

with R_i being the worst-case response time of Task τ_i and $hp(i)$ the subset of tasks which priorities are higher than the one of τ_i. The exact worst-case response time corresponds to the smallest fixed-point of Equation 2.

In classic scheduling results, preemptions were usually assumed to be performed at a 0 cost. However, this assumption does not hold anymore as soon as cache memories are considered as the additional delay incurred by preemptions can be as high as 40% of the task WCET [82]. So, the use of caches results in two main problems. First, predictability has to be ensured. As a result the CRPD has to be bounded. As stated in [25], there are several ways for assessing the cache interference penalty associated to every preemption. The penalty paid every time a preemption occurs can correspond to the time needed to refill either the entire cache, the lines accessed by the preempting task, the lines used by the preempted task, or finally the intersection of lines between the preempting and preempted tasks. But CRPDs can also threaten the system schedulability. To overcome this issue, CRPD bounds can be tightened (see Subsections III-C and V-A) or the CRPD itself can be reduced or even eliminated (see Section IV).

Note that, as shown in [86], classic scheduling policies such as RM, DM and EDF are no more sustainable as soon as CRPDs are considered. That means, for example, that a taskset might become unschedulable with RM, DM or EDF when decreasing a job execution time or even the delay incurred by a preemption.

D. Evaluation techniques

To evaluate the effectiveness of a technique and to compare it with previous ones, different approaches have been adopted in the literature. The evaluation can take place on a real hardware or, more frequently, analytically by using real benchmarks or randomly generated tasks.

a) Hardware: Different architectures, commonly used in embedded systems, are targeted: the ARM7 and 9[1] series such as in [53, 6, 5] or the MIPS R3000 as in [10]. In [77], Mezzetti and Vardanega prefer the LEON2[2] with the ORK+[3] real-time kernel, as it used in several projects from the European Space Agency (ESA)[4].

Typical cache characteristics, as found in [6, 89], are 512B to 32kB cache, either direct-mapped or way-associative, 16B line, a Block Reload Time (BRT) of 8 μs [6]. Consider for example a 4kB direct-mapped instruction cache with 16B line and assume the instruction size to be constant and equal to 4B. Then we have 256 cache sets, each of which can contain up to 4 instructions [52]. Note that, for real hardware, the number of sets is always a power of 2 [96].

b) Benchmark: Different benchmark tasks can be found in the literature. They are mainly intended for the WCET computation. The most common ones are probably the so-called Mlardalen Benchmarks[5] presented in [47] and used for example in [77, 6, 45, 53]. They consist in various programs such as binary search or data compression ones. They aim to represent typical code structures with nested loops or switch cases. Their sizes differ from a few bytes to several kB. PapaBench[6], used in [6, 9] is another benchmark which has the advantage to be based on a real real-time application [81]. Note that SCADE tasks, see [6], or applications coded in Ada, for example part of the Attitude and Orbit Control System of a real satellite studied in [44] as in [77], can also be used.

To compute WCETs using these benchmark tasks, an analysis tool is needed. Several are used throughout the literature, either free or commercial: aiT[7] in [77, 6, 53], Bound-T[8] in [77], Heptane[9] in [10, 89], Otawa[10]...

c) Taskset Generation: Most authors also use randomly created tasksets. The different task parameters can be generated the following way (see [6, 52]):

- task processor utilizations are generated using the UUnifast algorithm [75],
- periods are generated according to a log-uniform distribution (between for example 5ms and 500ms as in [52]),
- task cache usages are generated using the UUnifast algorithm,
- the number of reused blocks (see Section III-C) is generated according to a uniform distribution ranging from 0 to a fraction of the total number of memory blocks for the task.

When dealing with memory layouts (see Section IV-C), the position of these reused blocks also matters. So, in [69], the UUnifast algorithm is also used to generate a random distribution of these blocks throughout the tasks.

d) Monitored metrics: When dealing with cache and scheduling, different criteria can be studied. As stated in [6], it is impossible to study all possible combination between those parameters. The most commonly used variables are the BRT which is directly related to the preemption cost, the cache size, and, for the taskset, the total processor utilization, the number of tasks or their cache utilization (see [6, 52, 69]).

Usually, when dealing with WCETs and cache analysis, the hit/miss ratio [45], the WCET [3] or a bound on the CRPD [3] is measured or computed for each task to evaluate possible improvements brought by the considered method. When working with scheduling, either by improving analyses or devising new policies, the total processor utilization [21, 35], a schedulability ratio [16] or the weighted schedulability [52, 6] of the taskset is prefered. The weighted schedulability measure, introduced in [12], allows to reduce results to 2 dimensions without imposing a constant processor utilization:

$$W(p) = \frac{\sum_{\tau \in \mathfrak{T}} u(\tau) \cdot S(\tau, p)}{\sum_{\tau \in \mathfrak{T}} u(\tau)}$$

p being the studied parameter, $\mathfrak{T}$ a set of tasksets τ generated for equally spaced processor utilizations $u(\tau)$ and $S(\tau, p)$ the schedulable result (either 0 or 1) for Taskset τ under Parameter p.

E. Technique roadmap

Table II summarizes known techniques for exploiting cache memories in real-time predictable systems. These techniques will be detailed in the remaining of this paper.

In the table, techniques are classified depending on their focus. At one hand, techniques such as cache partitioning or locking aim at improving timing aspects by decreasing extrinsic and/or intrinsic cache interferences. At the other hand, cache-aware scheduling focus directly on improving the system schedulability by explicitly taking scheduling decisions depending on cache-related parameters.

III. Timing Analysis

As real-time scheduling focus on assuring timing requirements, the time needed for a task to complete must be known. But as depicted in Figure 2, this execution time is very dependant on the possible inputs for the task and on the hardware behaviour (pipeline and cache states for example). To ensure predictability, the worst-case execution time (WCET) of each task has to be considered.

A. General WCET computation issue

One way to get the WCET of a task is to use measurement-based methods, see for example [84]. But all possible execution paths have to be measured in order to get the longest one, which becomes difficult as soon as hardware features such as pipelines or caches are introduced. If not so, the measured execution time value might be an underestimation of the WCET which is unacceptable for hard real-time systems. As stated

[1] http://www.arm.com/products/processors/classic

[2] http://www.gaisler.com/index.php/products/processors

[3] http://www.dit.upm.es/~str/ork/

[4] http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors

[5] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[6] http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97

[7] http://www.absint.com/ait/

[8] http://www.bound-t.com/

[9] https://team.inria.fr/alf/software/heptane-static-wcet-estimation-tool/

[10] http://www.otawa.fr/

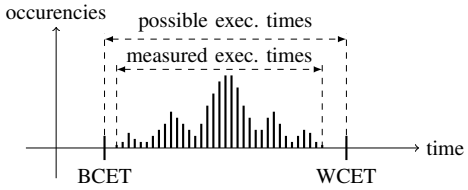


Figure 2. Possible execution times for a task

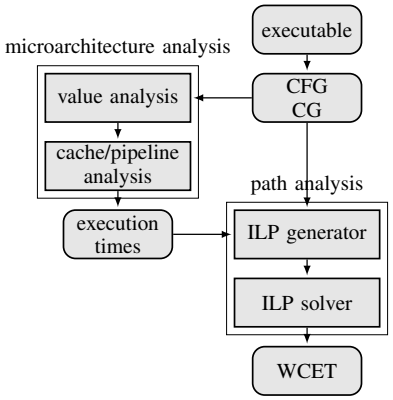


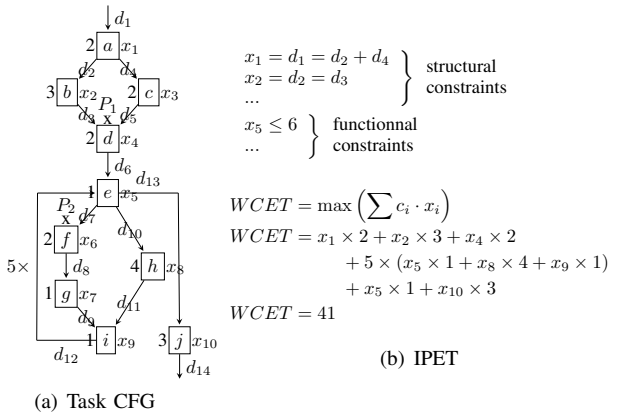
Figure 3. WCET estimation chain

in [90], it is almost always impossible to conduct exhaustive testing on a system.

So, static analysis is often used instead: the task code is analysed in combination with a model representing the hardware behaviour. As depicted in Figure 3, a low-level analysis determines the worst-execution times for the different program blocks of the task. Then, using the values computed at the previous step, a high-level analysis determines the longest execution path (WCEP: worst-case execution path) for the task. The WCET corresponds to the task execution time along this WCEP.

The low-level analysis computes execution time for each instruction. Because of complex architectures, the analysis has to deal with instructions overlapping because of pipelines, branch prediction, or caches. As a consequence, the whole task code has to be considered during the analysis. For pipeline analysis, see for example [108]. Work on branch prediction has been conducted in [122] and [23]. When dealing with caches, the easiest way would be to consider all accesses as misses. But this lead to highly pessimistic results, in particular for tasks with loops which are very commonly found in RTES. So, a cache analysis has to be conducted to determine which accesses will result in hits and which in misses. This analysis is presented in Section III-B.

Several high-level analyses exist to estimate the WCEP and so compute the WCET. They usually use the task Control Flow Graph (CFG), in which blocks correspond to instruction sequences without conditional jumps and edges to control flows, and Call Graph (CG), in which blocks correspond to functions and edges to function invocations. In [122], Colin and Puaut propose a tree-based analysis whereas in [61], later extended



(a) Task CFG

(b) IPET

Figure 4. Example of WCET computation using IPET

for example in [42] and [99], an integer linear programming technique, called Implicit Path Enumeration Technique (IPET), is introduced. It uses the following objective function:

$$\text{WCET} = \max\left(\sum_i c_i \cdot x_i\right)$$

where c_i stands for the execution time of basic block i and x_i is the number of times the block is executed. Several constraints are added to represent structural aspects (incoming and outcoming edges in the task CFG) and functional ones (loop iteration bounds, mutual exclusive paths). See Figure 4 for an example of a simplified WCET computation using IPET. IPET is often implemented in WCET analysis tools, such as aiT[11].

For more details on WCET computation methods and on existing tools, refer to the survey by Wilhelm *et al.* [120].

B. WCET cache analysis

To compute a tighter WCET, a cache analysis is conducted to categorize each reference, as proposed in [49], in particular Always Hit (*AH*) references, i.e. accessed memory blocks that are already into the cache and for which a later access will incur no additional cost. To do so, different cache analyses have been developed, mainly for instruction caches which are easier to study. In [109], Mueller proposes to use static cache simulation, whereas in [39], Ferdinand and Wilhelm prefer to use abstract interpretation.

Data caches have been studied for example in [39] and [92]. But the analysis is more complex as potential write operations to the cache (and then to the main memory) might occur, see [39]. Moreover, data addresses cannot always be determined statically, see [118].

Hereafter we focus on the cache analyses presented in [39]. Ferdinand and Wilhelm use the concept of *Abstract Cache State* to represent the potential cache content at a given program point. They introduce three fixpoint analyses: the *must*, *may* and *persistence* analyses to categorize the different

[11] http://www.absint.com/ait/

instructions of a program. For each program point (just before reaching a CFG node), the abstract cache state is computed by merging all incoming cache states using a *join* function. Then the cache state is *updated* (which corresponds to the memory accesses made by the task at that particular node) to get the outcoming cache state for the node.

The *must* analysis is use to determine the AH references: at each node of the CFG, the cache *must* content, i.e. memory blocks that are sure to be in the cache at that program point, is computed. So the *join* function only keep memory blocks which belong to both input cache states. Consider the CFG depicted in Figure 4(a) and a 4-line direct-mapped cache. At Program Point P_1, the abstract cache state is: $[\{a\}, \{\}, \{\}, \{\}]$, which means that only Memory Block a is sure to be in the cache at that point. Indeed, accesses to b or c depend on the path which has been taken.

The *may* analysis allows to determine the AM references: the cache *may* content corresponds to the blocks that may have been accessed before the considered program point and may not have been thrashed from the cache. At Program Point P_1 we have: $[\{a\}, \{b\}, \{c\}, \{\}]$. The AM references are computed by taking the complement of the may content. References that are in the may content but not in the must one are said to be Non-Classified. Usually, to upper-bound the WCET, these references are changed to AM.

Finally, Ferdinand and Wilhelm introduce a *persistence* analysis to decrease the cache analysis pessimism. It allows to classifiy references as First Miss(FM): the first access to the reference may result in a miss but all further accesses are assured to be hits. Consider Program Point P_2 in Figure 4(a). The corresponding must cache content is $[\{\}, \{\}, \{\}, \{\}]$ because of the mutual exclusive paths with compose a loop iteration. So f would be classify as AM. However, once it has been accessed (if he is ever accessed) it cannot be removed from the cache, as neither g, e, d nor h maps to the same cache line. So, thanks to the persistence analysis, f is classified as FM.

For more details about cache analyses and how they are generalized to set- and fully-associative caches, see Ferdinand and Wilhelm [39]. Note also that a refinement of the cache analysis using abstract interpretation has been proposed in [100]: using model checking some NC accesses can be identified as AH decreasing the analysis pessimism.

C. Including Cache-Related Preemption Delays into the WCET

However, one issue remains: how to account for CRPDs? The WCET computed using the methods described in the previous two sections corresponds to the worst-case execution time of a task executing on its own without any external interference. But of course, as soon as preemptive scheduling is considered, we have to account for potential damage done to the cache by preempting tasks.

Once more, the easiest way to compute a WCET taking preemption costs into account is to assume a cache miss at each reference. This is a very pessimistic but general approach, as it is only dependant on the task being analysed (so the computed WCET can be used for several task systems scheduled with different algorithms).

A less-pessimistic way is to increase the WCET to account for the CRPDs. In [116], Ward *et al.* distinguish between *preemption-centric* and *task-centric* methods.

Preemption-centric methods add an upper-bound on the CRPD, δ_i, to the preempting task, as in [13]:

$$\hat{C}_i = C_i + \delta_i$$

δ_i can be computed as the cost of reloading the entire cache, or less pessimistic, as the damage the preempting task can have on the cache content. This interference can be modelled using the notion of Evicting Cache Block (ECB), introduced in [110]. For direct-mapped caches, there are as much ECBs as cache lines that may be accessed by the task during its execution. The upper-bound comes straight-forward:

$$\delta_j = BRT \cdot |ECB_j| \tag{3}$$

For task-centric methods as in [102] and [28], the WCET is increased by an upper-bound on the CRPD for one preemption, γ_i, multiplied by an upper-bound on the number of possible preemptions, n, the task may suffer:

$$\hat{C}_i = C_i + n_i \cdot \gamma_i$$

To compute the upper-bound on the number of preemptions several methods have been proposed, depending on the scheduling policy. As stated in [116], for FTP scheduling, a simple bound is given by: $n_i = \sum_{j=1}^{i-1} \left\lceil \frac{T_i}{T_j} \right\rceil$. However, such a bound overestimates the number of potential preemptions. So more precise computations have been proposed in [94] and in [38] where upper-bounds on the number of preemptions for both FTP and EDF scheduling policies are proposed.

As presented in [58], Useful Cache Blocks (UCBs), which correspond to the reuse of the available cache contents by the preempted task, can be used to compute γ_i. Consider the example depicted in Figure 4(a): at Program Point P_2, Memory Blocks e, f, g and h may have been accessed by T. f, g and h are UCBs as their access will not result in additional reloads, if no preemption is considered, as they are already in the cache. But e is no UCB as Block i will evict it from the cache before it is re-referenced. Then, the upper-bound is simply computed as:

$$\gamma_i = BRT \cdot |UCB_i| \tag{4}$$

Lee *et al.* propose in [58] a method to compute the UCBs of a task using a representation of the cache contents. Such a representation was enhanced by Negi *et al.* in [80] but at a higher computing complexity. So Staschulat and Ernst introduce in [104] a trade-off between the above two methods. A different but less frequently used approach is proposed in [92] for data caches, using access patterns.

To compute a tighter CPRD bound for the task-centric approach, a combination of ECBs and UCBs can be considered as proposed in [80] and [101]:

$$\gamma_{i,j} = BRT \cdot |UCB_i \cap ECB_j| \tag{5}$$

To improve the CRPD bound computation, Altmeyer *et al.* also introduce in [3] the notion of Definitely-Cached Useful

Block (DC-UCB): it allows tighter results in comparison with UCBs because it only accounts for cache misses which have not already been considered during the WCET analysis. As a consequence, this method is only safe when used in combination with a WCET bound. Note that, as soon as fully- or set-associative caches are considered, and as stated in [22], UCBs and ECBs can be used only with the LRU cache replacement policy. For CRPD bound computations with other cache policies such as FIFO, the notion of relative competitiveness, presented in [97], can be used.

According to [116], task-centric methods are highly pessimistic when the number of tasks is high (because many possible preemptions have to be considered). On the other side, preemption-centric methods become highly pessimistic when task WSSs are highly variant. As a consequence, Ward *et al.* propose in [116] a mixed-approach. The CRPD is accounted for both the preempted and preempting tasks:

$$\hat{C}_i = C_i + n \cdot \max(0, \gamma_i - G) + G$$

G is computed using linear programming in order to minimize the taskset total processor utilization.

IV. Memory management

The idea is here to reduce and/or eliminate the extrinsic and/or intrinsic cache interferences by playing on the cache mapping. Decreasing or even removing these cache side-effects can be intended in order to: minimize a given task WCET, decrease the overall processor utilization or maximize the system schedulability.

Different techniques exist to achieve such goals. The most common ones are cache partitioning, cache locking and task layout which will be presented hereafter. However, other methods can be found in the literature. In [119], Whitham *et al.* propose to save the cache state on a stack, whenever a task is preempted, and restore it when the task resumes its execution, whereas in [2], Allard *et al.* prefer to divide the cache into two layers: one is used as an usual cache while the second one saves its content to the main memory or restore a previous content from the main memory. A similar approach is used in [117] along with a non-preemptive fixed-priority scheduling algorithm. Finally, in [95], Reineke et al. propose to implement a new cache replacement policy which takes into account the task to which the cached memory block belongs. This policy allows to decrease the CRPD by avoiding some block reloads.

The different methods listed here are synthesized in Table II.

A. Cache partitioning

Cache partitioning aims to eliminate potential inter-task cache conflicts as they are source of unpredictability. The cache is divided into several sets which might be of different size. Tasks are then assigned to those partitions and so cannot interfere with one another. Note that, very often, a common partition is added for non-critical tasks or shared data, see for example [56]. The main question is, of course, to find the number of partitions and their respective sizes.

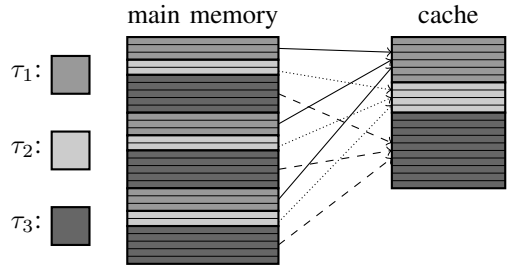


Figure 5. Example of software cache partitioning for three tasks τ_1, τ_2 and τ_3.

Cache partitioning can either be implemented at the hardware level, as proposed in [56] by modifying the memory management unit behaviour, or be software-based, as first introduced in [121] and improved in [79]. Recently, software-based partitioning has been mostly considered because, as stated in [79], the hardware-based one has several drawbacks: the partition sizes are fixed beforehand and custom-made hardware architectures have to be used while software-based partitioning can be applied directly to all off-the-shelf architectures. To implement software-based partitioning, OS-controlled techniques to manage the cache can be used as presented in [62], or code modifications can be introduced at the compiler (linker) level as proposed in [79]. This latter technique is based on the fact that the location of task memory reference in the cache is determined by its position in the main memory. So, by changing properly task reference locations in the main memory, as depicted in Figure 5, then those references will map only to a restrictive number of cache lines creating a partition. But, as the task code may be split, unconditional jumps might be added resulting in a potential increase in the task WCET. Note that it is also the case for hardware-partitioning, as it needs additional circuits to be implemented, as shown in [57]. As stated in [114], hardware partitioning is usually way-based, whereas software partitioning is typically set-based.

By comparing cache partitioning with other methods (in particular task layout), Altmeyer *et al.* identify in [6] partitioning to be more suited for tasks with short WCETs and periods as, in this case, CRPDs might be very high in comparison with their execution times. Such tasks can typically be control-oriented applications.

a) Fully-partitioning: To fully eliminate extrinsic interference, private partitions are used: all tasks are isolated from one another. The simplest way to compute a task partition size is to consider the task size relative to the taskset size as proposed in [79]. But, because tasks have access to a smaller amount of cache memory, their WCET may increase and thus threaten the system schedulability. To overcome this problem, Plazar *et al.* propose in [87] to base the partition size selection on the goal of minimizing the overall system processor utilization. However, as stated in [6], minimizing the processor utilization does not necessarily lead to optimality in terms of schedulability for the system. So, Altmeyer *et al.* focus on a partitioning algorithm aiming at maximizing

directly the system schedulability instead of minimizing the total processor utilization [6].

b) Hybrid-partitioning: The techniques presented above allow to eliminate extrinsic cache interferences but at the cost of potentially increasing the intrinsic interference which may have dreadful consequences on the system schedulability. Indeed, very often, as shown for example in [6], the increased predictability provided by cache partitioning (as no CRPD has to be accounted for any more) does not compensate for the performance degradation in WCETs. So, Busquets *et al.* propose in [107] a hybrid partitioning technique: some critical tasks will have to share a same partition. They propose a task partition assignment under RM scheduling: highest priority tasks will be assigned to same-sized private partitions, whereas the remaining tasks with lower priorities will share one common partition. They show that hybrid-partitioning allows to achieve better processor utilization than fully-partitioning techniques when the cache gets smaller. Note that, for large caches, both approaches perform quite identically. In [21], Bui *et al.* release the partition constant size constraint. Partition assignment is stated as an optimization problem which goal is to minimize the total processor utilization. However, they prove such a problem to be NP-hard as it can be reduced from the knapsack problem in polynomial time. So, they use a genetic algorithm to compute a number of partitions with different sizes and the corresponding task assignment. In [106], Tan and Mooney propose a hardware-based partitioning solution using additional registers for fixed-priority tasks, called prioritized cache. Priorities are assigned to cache partitions such that only tasks with priorities higher or equal to the one of the partition have access to it.

B. Cache locking

Achieving predictability when using caches becomes difficult because of intrinsic and extrinsic cache interferences. It is often hard to know precisely the cache contents at a given instant. Full partitioning allows to eliminate the extrinsic interference but at the cost of potentially decreasing the schedulability. Another solution is to use cache locking: some cache lines are prevented from being overwritten once some content has been loaded into them. This is done through hardware mechanisms that allow to control the cache contents at the software level. Such mechanisms are implemented in off-the-shelf architectures such as the ARM9 series[12] (see [53]). Note that most locking techniques deal only with instruction caches. A focus on data caches is given in [113]. To tell the hardware which content to lock, either debug registers can be used as in [10] or lock/unlock instructions have to be added in the task code as in [91].

Cache locking can be *static*, i.e. the cache locked content does not change during the whole system execution, or *dynamic*, i.e. the locked cache content can change at runtime. Consider the example depicted in Figure 6. Memory references are denoted by letters. In 6(a), τ_1's and τ_2's worst-case execution paths are depicted by the sequence of their memory

[12]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0092b/I14301.html

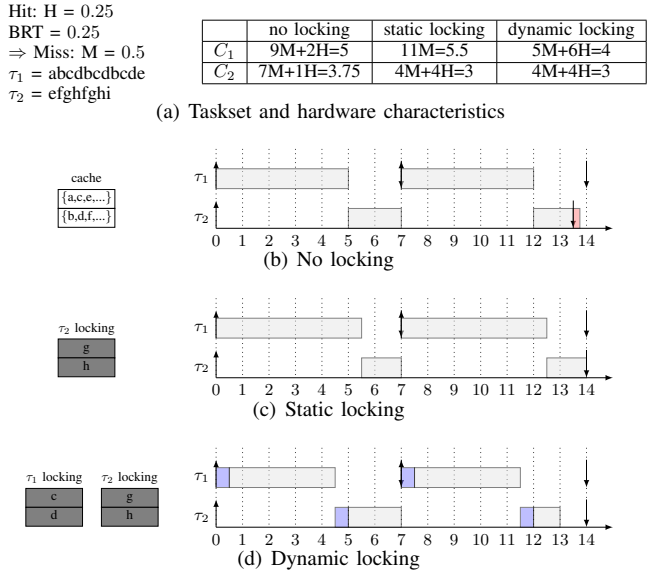


Figure 6. Example of different locking techniques. The up arrows represent the job releases and the down arrows the task deadlines.

accesses. A cache hit is supposed to result in an execution time of 0.25 whereas a miss leads to an additional delay of 0.25. Tasks are scheduling under Rate Monotonic. When no locking technique is used, τ_1's second job preempting τ_2 incurs an additional delay causing τ_2 to miss its deadline. In 6(c), static locking is used: Blocks g and h belonging to τ_2 are loaded in the cache at the system start-up and locked. As a consequence, τ_1 cannot use the cache and its WCET is increased, see the table depicted in 6(a). But no CRPD is incurred anymore and as a result the system becomes schedulable. Using dynamic locking as shown in 6(d), each time τ_1 (respectively τ_2) is released or resumes its execution, Blocks c and d (resp. g and h) are loaded into the cache. It allows smaller WCETs, see the table in 6(a), and, once more, the system is schedulable. We distinguish hereafter between full locking techniques, where the whole cache is locked at each instant, and partial ones, where some lines are left unlocked.

a) Full Locking: Campoy *et al.* in [31] are the first to propose the use of cache locking to improve predictability. They use global static locking, i.e. at every moment each task owns a portion of the cache. Blocks to be locked are chosen according to a genetic algorithm which tries to minimize the average task response times. In [41], Falk *et al.* prefer to focus on minimizing the WCET. The proposed algorithms are only heuristics as the static locking problem aiming to minimize the WCET is NP-hard, as proved by Liu *et al.* in [64]. In [30], Campoy *et al.* find that static locking is more suitable to achieve higher predictability, but, in most cases, dynamic locking shows better performances. So, in [10], Arnaud and Puaut prefer to use local dynamic locking, i.e. at each instant, a task owns the whole cache. Tasks are split into sets of basic blocks using a greedy algorithm which tries to minimize each task WCET. For each set the locked cache state is known statically. Finally, in [88] and latter in [89], Puaut

and Pais introduce an algorithm to select the reload points using a cost function based on the goal of reducing the WCET. Then, memory contents to be locked at each of the selected reload points are chosen. In [53], Liu *et al.* use a tree-based approach to represent tasks. They propose static, semi-dynamic and dynamic locking algorithms aiming to minimize the total processor utilisation. They show further processor utilization reduction compared to previous algorithms focusing only on the WCET, such as in [41].

b) Partial Locking: But as for cache partitioning, global locking techniques, either static or dynamic, result in tasks having accessed to a limited cache space which may in turn increase their WCETs and thus threaten the system schedulability. So, in [35], Ding *et al.* propose to use partial locking: one portion of the cache is statically locked for each task while a common portion of the cache is left unlocked. Locked contents are selected according to a cost-benefit analysis: the aim is to minimize the worst-case response times. In [35], Ding *et al.* focus on static locking, but they extend their work to dynamic locking in [36] showing better results.

c) Locking with partitioning: Contrary to the authors cited before, Vera *et al.* study locking for data caches in [113]. They propose a combined approach: cache partitioning is used to eliminate inter-task (extrinsic) interference whereas cache locking is aimed at insuring intra-task (intrinsic) interference predictability.

C. Memory layout

As stated for software-based cache partitioning, the position of a memory reference in the main memory influences its location in the cache. Memory layout techniques focus on reducing either the intrinsic or the extrinsic cache interference. *Code positioning* (respectively *task placement*) aims to reduce intra-task (resp. inter-task) conflicts by modifying, during the compilation process (resp. when loading the tasks into the main memory), the position of code sections of a task (resp. of the entire task) but without necessary creating cache partitions. Figure 7 depicts an example of task placement taken from [45]. Tasks τ_1 and τ_3 have short periods whereas τ_2 has a larger one. So τ_1 and τ_3 may conflict with each other very frequently causing high CRPDs. As a result, the taskset is not schedulable as depicted in 7(a). Changing task positions in the main memory as in 7(b) allows to have τ_1 and τ_3 mapping to different cache lines. τ_1 preempting τ_2 does not incur CRPDs anymore. So the total inter-task interference is significantly reduced and the taskset becomes schedulable.

In [68], Luniss *et al.* show that, for FTP scheduling, task positioning can allow similar processor utilization as EDF. Moreover, Altmeyer *et al.* find in [6] that, in most cases, the use of a task layout, resulting from the algorithm proposed in [69], leads to better results than cache partitioning in terms of the number of schedulable tasksets. However, as soon as task positioning is considered, changes in the taskset or in the scheduling policy implies to recompute layouts, so possibly modifying the task WCETs.

a) Code positioning: In [111], Tomiyama and Yasuura focus on code placement techniques to decrease the task

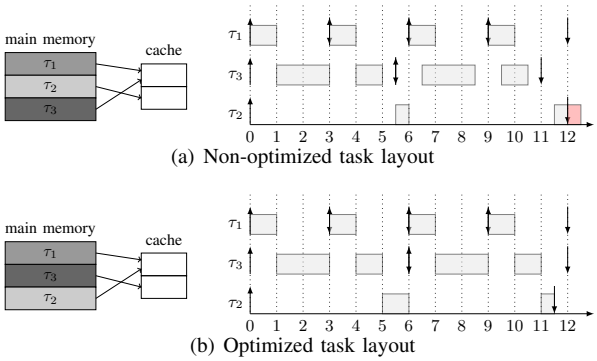


Figure 7. Example of different layouts for Tasks $\tau_1(1,3,3)$, $\tau_2(1.5,12,12)$ and $\tau_3(3,6,6)$ scheduled using Rate Monotonic.

miss rate and so achieve intra-task conflict reduction. In [76], Kowarschik and Weiss propose several code modification techniques such as loop interchange, as well as data layout optimization in order to increase locality and so reduce cache misses. In [65], Lokuciejewski *et al.* propose three positioning algorithms to reduce intra-task conflicts. They focus on high call frequency procedures which they try to allocate contiguously in memory. In particular, they use a greedy algorithm and a heuristic one to achieve such a goal. As a consequence, WCETs are reduced. In [40], Funk and Kotthaus propose a cache-aware code positioning optimization driven by WCET information based on conflict graphs to determine potential intrinsic conflicts. Tasks are split in fragments and a greedy-approach-based heuristic is used to position the different fragments in memory. At each step of the algorithm, a new WCET is computed and compared to the previous one, in order to know if any improvement has been achieved. Because tasks are split, code modifications have to be introduced such as unconditional jumps. In [77], Mezzetti and Vardanega focus on the problem of incremental development for industrial needs.

b) Task placement: In [45], Gebhard and Altmeyer prefer to study task placement to decrease inter-task conflicts. Their idea is to maximize the number of persistent cache sets to allow more precise WCET estimations for preemptively scheduled tasks. To find a optimal task layout, they consider an optimization problem using an integer linear program (ILP) formulation. The objective function can be basically seen as the sum over the cache conflicts for each task. But, Gebhard and Altmeyer show that such a problem is unfortunately NP-complete. So, they propose a simulated annealing algorithm as a heuristic approach. Lunniss *et al.* extend this algorithm in [69] by using UCBs and ECBs to determine whether an evicted block will need to be reloaded or not. Later, in [8], Altmeyer et Gebhard derive a metric to compare different memory layouts. Then, they approximate an optimal layout with respect to this metric and memory accesses are classified as persistent or endangered. Eventually, they compute a safe bound on the WCET, thanks to that classification.

V. Enhanced task models

An other way to deal with cache issues is to work at the scheduling level. Thus, two main orientations can be followed:

1) the first one is to ensure predictability by incorporating the CRPDs in the schedulability analysis: in contrast to Section III, the CRPD and the WCET are considered here apart from each other,
2) the second one is to increase the system schedulability by reducing the number of preemptions through scheduling modifications or by explicitly considering cache-related preemption delays when taking scheduling decisions.

The different methods are synthesized in Table II.

A. Cache-aware scheduling analysis

In Section III, preemption delays were accounted for into the WCET. But such techniques often result in very pessimistic WCETs, because the number of preemptions is hard to determine accurately. To overcome this shortcoming, the CRPD can be considered apart from the WCET. The idea is first to compute an upper-bound on the CRPD due to one preemption from a higher priority task, by considering the preempted and/or preempting tasks. Then these costs are incorporated into the schedulability analysis, by extending the classic response time analysis (RTA) for FTP scheduling, as proposed by Busquets-Mataix *et al.* in [25]:

$$\hat{R}_i = \hat{C}_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{\hat{R}_i}{T_j} \right\rceil \cdot (\hat{C}_j + \gamma_{i,j}) \tag{6}$$

$\gamma_{i,j}$ represents a bound on the CRPD experienced by Task τ_i each time it is preempted by a higher priority Task τ_j. Note that $\hat{C}_i$ is the WCET of the task considered on its own, i.e. without taking into account possible delays due to other tasks, contrary to the traditional C_i as introduced in II-C.

a) Preempting task: Busquets-Mataix *et al.* [24], and later Tomiyama and Dutt in [110], focus on the preempting task to bound the CRPD. As in Section III, Evicting Cache Blocks, i.e. cache lines that may be accessed by the task during its execution, are used to represent the potential damage the preempting task can have on the cache. The ECB-only approach uses the same ECB-bound (3) as in Section III:

$$\gamma_j^{ecb} = BRT \cdot |ECB_j| \tag{7}$$

Consider the example depicted in Figure 8. τ_1 is assumed to have a higher priority than τ_2 which in turn has a higher priority than τ_3. If CRPDs were not taken into account, then the taskset would be considered schedulable, according to the Rate Monotonic Analysis (RMA) presented in [63]: $U = \sum_1^3 \frac{C_i}{T_i} \leq 3 \times (2^{1/3} - 1)$. However, using Formula 3, τ_1 preempting τ_3 (respectively τ_2 preempting τ_3) incurs an additional delay $\gamma_1 = 1 \times 2 = 2$ (resp. $\gamma_2 = 1 \times 3 = 3$). As a result, using Formula 6, the worst-case response time of τ_1 is $\hat{R}_3 = \hat{C}_3 + (\hat{C}_1 + \gamma_1) + (\hat{C}_2 + \gamma_2) = 12$ and the task misses its deadline.

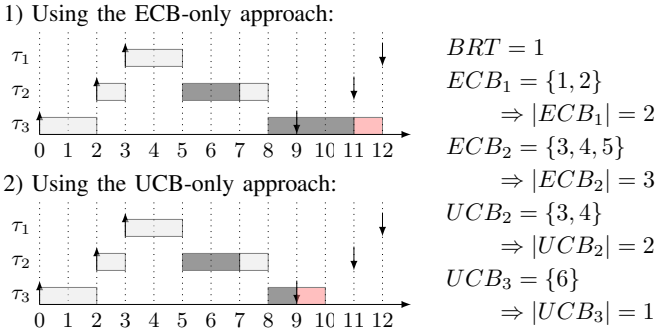


Figure 8. Schedules for tasks $\tau_1(2, 9, 9)$, $\tau_2(2, 9, 9)$ and $\tau_3(3, 9, 9)$ using the ECB- (upper figure) and UCB-only (lower figure) approaches to bound the CRPD (numbers within {}-brackets for ECBs and UCBs correspond to cache line indexes).

Busquets-Mataix *et al.* show in [24] that, when using the CRPD bound given by Equation 3, the RTA computed using Equation 6 clearly outperforms the cached version of the RMA schedulability test presented in [13] which uses a WCET including the CRPDs.

b) Preempted task: In [58], Lee *et al.* prefer to focus on the preempted task to bound the CRPD, using Useful Cache Blocks (UCBs). We recall that UCBs correspond to the reuse of the available cache contents by the preempted task. But the simple bound given by Equation 4, presented in Section III, cannot be used here. This is because of the impact of potential nested preemptions. Consider again the three tasks depicted in Figure 8. Applying Formula 4, we get: $\gamma_2 = 1 \times 2 = 2$ and $\gamma_3 = 1 \times 1 = 1$. Using the RTA given by Equation 6, we see that τ_3 can suffer as most one preemption from each higher priority task. Adding twice the CRPD γ_3 will result in $\hat{R}_3 = \hat{C}_3 + (\hat{C}_1 + \gamma_3) + (\hat{C}_2 + \gamma_3) = 9$. However, τ_1 preempting τ_2 results in a preemption cost γ_2 of 2 and thus, as shown in Figure 8, the worst-case response time of τ_3 becomes 10 causing the task to miss its deadline. So intermediate tasks have to be considered to account for potential nested preemptions. Finally, the CRPD incurred by a higher priority Task τ_j on Task τ_i can be computed using the UCB-only approach:

$$\gamma_{i,j}^{ucb} = BRT \cdot \max_{\forall k \in hep(i) \cap lp(j)} \{|UCB_k|\} \tag{8}$$

with $hep(i)$ the set of tasks of priority higher or equal to τ_i and $lp(j)$ the set of tasks of lower priority than τ_j.

Applying this formula on the example depicted in Figure 8, we get: $\gamma_{3,1}^{ucb} = 1 \times \max\{2, 1\} = 2$ and $\gamma_{3,2}^{ucb} = 1 \times \max\{1\} = 1$, and so $\hat{R}_3 = 10$.

c) Both tasks: But, considering either the preempting task or the preempted task on their own is very pessimistic: it is possible that those tasks do not conflict into the cache. Consider again the example depicted in Figure 8 and a 8-lines direct-mapped cache. $ECB_1 \cap UCB_2 = \emptyset$, $ECB_1 \cap UCB_3 = \emptyset$ and $ECB_2 \cap UCB_3 = \emptyset$: so, τ_1 preempting τ_2 or τ_3 and τ_2 preempting τ_3 will not cause additional delay as evicted cache lines are not used later on by the preempted tasks. So UCBs and ECBs can be combined to decrease this source of pessimism and so tighten the CRPD bound. Different

approaches have been proposed in the literature such as UCB-Union in [105] and ECB-Union in [5]. They are improved in [52] to estimate more accurately the number of preemptions in the schedulability analysis. A comparison of these different approaches can be found in [52].

Finally, all these results are extended to EDF in [67]. In [68], Lunniss *et al.* compare FTP scheduling and EDF as soon as CRPD is considered. They show that EDF still offers better performances than FTP scheduling, but the gap is narrower than for scheduling without CRPDs. Note that CRPD accounting techniques based on UCBs and ECBs have also been proposed for more complex scheduling paradigms, such as hierarchical scheduling [71, 70].

B. Limited preemption scheduling

The methods presented in the previous section only consider the CRPD to assure predictability. No change was made to the scheduling policy itself. We now focus on techniques to increase the system schedulability by controlling the preemptions. Indeed, as stated in [115] and illustrated in Figure 9, for FTP scheduling, preemptive and non-preemptive schedulings (respectively denoted FPPS and FPNS) are incomparable. Moreover, using limited-preemption scheduling can make a system, unschedulable under both preemptive and non-preemptive schedulings, schedulable, as depicted in Figure 10 using preemption thresholds on the same example as in Figure 9.

Historically, these methods only aim to increase schedulability by controlling preemptions without considering any preemption cost. However, recent works, such as [20], have focused on combination between classic limited-preemption techniques and CRPD-aware schedulability analyses.

We focus on two limited-preemption scheduling categories: fixed-task preemption thresholds scheduling (FPTS) and fixed-task deferred preemption scheduling (FPDS). For FPDS, either the floating-Non Preemptive Region model (f-NPR) or the fixed-NPR one, also referred to as the Fixed Preemptive Points model (FPP), can be used. We mainly focus on the latter as the f-NPR model is poorly suited to take CRPD into account. A generalization of both FPTS and FPDS has been introduced in [19] by Bril *et al.*. Note that there also exist other techniques such as the one introduced by Dobrin and Fohler in [37]: instead of modifying classical preemptive-scheduling, tasks attributes are changed to reduce the number of preemptions.

a) Preemption Thresholds: The notion of preemption thresholds was first introduced in the ThreadX RTOS[13] and later theorized by [115]. Alongside their priority, each task is given a preemption threshold θ_i. A task can only be preempted by a higher priority task which priority is also higher than the lower-priority task threshold. An example of this policy, applied to the example presented in Figure 9, is depicted in Figure 10. A Time 10, a new job of τ_2 is released. τ_2 has a higher priority than the running task τ_3, but as its threshold is not higher than τ_3's priority, no preemption can occur at this point, and, as a consequence, τ_3 can finish its execution and so meets its deadline.

[13]http://rtos.com/products/threadx

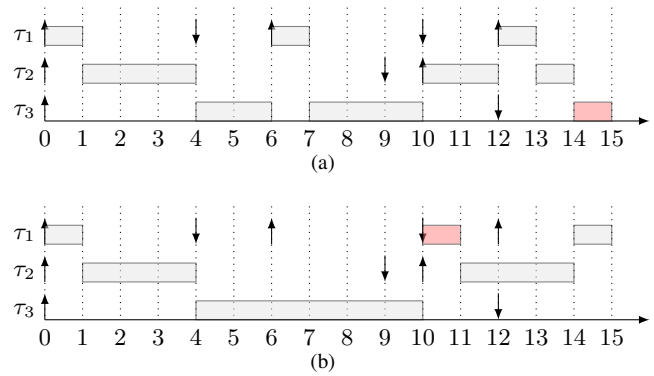


Figure 9. Non dominance of FPPS and FPNS taken from [26] for taskset example: $\tau_1(1,4,6)$, $\tau_2(3,9,10)$ and $\tau_3(6,12,18)$ ($\tau_i(C_i,D_i,T_i)$).

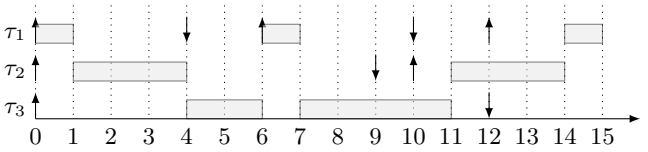


Figure 10. Example of FPTS taken from [26] on the same taskset as in Figure 9; with preemption thresholds: $\theta_1 = \theta_2 = 3$ and $\theta_3 = 2$.

In [20], Bril *et al.* introduce a schedulability analysis for preemption threshold scheduling which incorporates CRPD. They combine works on preemption thresholds without CRPD made by [55] with the ECB- and UCB-based approaches proposed amongst others in [52].

b) floating-Non Preemptive Region: Here the idea is that each task has a maximum interval of time, called a Non-Preemptive Region (NPR) of size q_i, during which it cannot be preempted by any other task. Under the f-NPR model, when a higher priority task is ready, the task already executing will only be preempted after q_i units of time. f-NPR scheduling has been studied for example in [11], [15] and [43]. Some works have also been conducted to incorporate the CRPD in the f-NPR model. In [93], Ramaprasad and Mueller work on bounding the worst-case response time as soon as data caches are considered. In [73], later improved in [72], Marinho *et al.* compute an upper-bound on the CRPD to be included into the WCET based on a preemption delay function. This function represents the preemption cost tied with program-execution progression. But as the NPR is floating, it is nearly impossible to take CRPD into account to decide when to preempt or not.

c) Fixed Preemptive Points: Under the FPP model, each task job is divided into m_i non-preemptive subjobs of size $q_{i,k} \leq q_i$. A task can only be preempted between two consecutive subjobs: for example, as depicted in Figure 11, τ_1's preemption on τ_3 is delayed until time 8 which corresponds to the end of τ_3's first subjob. The FPP model makes it possible to protect some code sections (small loops, or sections with accesses to shared resources) by including them in non-preemptable subjobs. However, implementing preemption points is not easy and requires task code modifications. This problem is discussed in [51]. In addition NPR sizes have

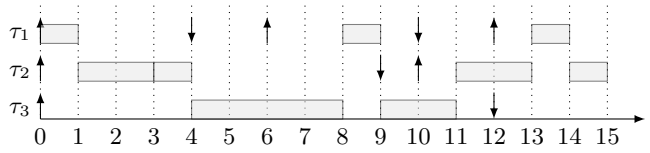


Figure 11. Example of FPP taken from [26] on the same taskset as in Figure 9, with τ_2 split in two subjobs of size 2 and 1 and τ_3 split in two subjobs of size 4 and 2.

to be recomputed as soon as the taskset changes, and, as a consequence, preemption point placement in the code may have to be changed. Scheduling with Fixed Preemptive Points has been studied in particular in [18], [123], [124] and [43].

In [4], Altmeyer *et al.* take CRPD into account to compute the maximum blocking time a task can suffer because of a non-preemptive subjob of a lower priority task. Indeed, because of the preemption cost paid when a task resumes its execution, the next preemption point may be delayed and, as a consequence, the blocking time is increased.

C. Cache-aware scheduling

The methods presented in the previous subsections aim to increase the system schedulability by controlling the preemptions. CRPDs are only accounted for (or not) during the schedulability analysis to ensure predictability. However, decreasing the number of preemptions do not necessarily decrease the total CRPD as shown in [16]. So, to overcome this issue, the CRPDs have to be considered when taking scheduling decisions.

The cache-aware scheduling problem can be tackled by modifying the pre-processing step of different classic scheduling decisions. For example, choosing the task priorities or fixed preemption points based on cache consideration, will have an effect of the decision of scheduling a job at a given instant. But as the influence of cache issues on the scheduling decisions is only indirect, these solutions might not be optimal. So the problem of devising scheduling policies that are based directly on cache parameters has to be investigated.

a) CRPD-aware pre-processing: In [112], Tran *et al.* propose to extend Audsley's algorithm for assigning task priorities to take CRPDs into account. The basic idea is to test at each priority assignment step the system schedulability using the cache-aware RTA. In [20], Bril *et al.* use their schedulability analysis for the preemption threshold policy to propose an optimal algorithm to assign preemption thresholds which aims to minimize the CRPD. In [114], Wang *et al.* combine preemption threshold scheduling and partitioning. A same threshold is given to tasks assigned to a same cache partition such that they cannot preempt each other. As a consequence, no CRPD is incurred. The partition and threshold assignment problem is formulated as an Integer Linear Program. A heuristic algorithm is also proposed. The CRPD can also be considered when selecting the preemption points. In [103], Simonson and Patel propose to split tasks at points having the minimum number of live cache lines while ensuring that the longest non-preemptive interval for the task is not exceeded. The notion of live cache lines correspond to what Lee *et al.* call, in [59], UCBs. This method allows to reduce the total preemption overhead. However, as stated in [32], it does not compute a globally optimal solution. Finally, in [17], later improved in [16], Bertogna *et al.* introduce an optimal preemption point placement algorithm to minimize the total preemption overhead rather than necessary minimizing the number of preemption points. The CRPD is then added into the task WCET (as the number of preemption points and their respective costs are known). In [83], Peng *et al.* extend this work to task codes with conditional branches. One drawback of these works is, according to [60], the possible overestimation of the number of preemptions: when computing the task WCET accounting for CRPD, all preemption points are assumed to result in a preemption. In [32], Cavicchio *et al.* introduce a more accurate CRPD calculation by considering the cache blocks that are reloaded during two consecutive preemption points. As a consequence, it allows more tasksets to be schedulable in comparison with the Bertogna *et al.*'s method, as the total preemption overhead is significantly decreased.

b) Optimal cache-aware scheduling: We now consider the more general problem of cache-aware scheduling, i.e. taking optimal scheduling decisions according to cache issues. Cache-aware scheduling can be tackled from two different angles:

- CRPD-*aware* scheduling: decisions are based on CRPD information, the scheduler might try to minimize them in order to increase schedulability.
- *Cache-aware* scheduling: decisions are based on cache utilization information, the scheduler might try to maximize cache re-utilization by the tasks, for example if external libraries or functions are shared among tasks.

Unfortunately, as stated in [85], both problems are NP-hard in the strong sense for uniprocessor systems.

To the best of our knowledge, little research has been conducted on optimal uniprocessor cache-aware scheduling. However, some work exists as far as multiprocessors are concerned. But most of them deal with partitioned multiprocessor soft-real time scheduling: cache-aware decisions influence only the taskset partitioning process in order to reduce conflicts between tasks, see for example in [29] and [46].

In [86], an offline solution to the CRPD-aware scheduling problem is proposed for a task model with a constant CRPD bound for each task, using a Mixed-Integer Linear Program (MILP) formulation. The computed solution provides a comparative point to evaluate the loss of schedulability of classic scheduling policies such as EDF as soon as CRPDs are considered. The model proposed in [86] is pessimistic as it considers only a preemption cost per task, meaning for example that each task has to reload all of its UCBs after every preemption. But extending this model to better model the CRPD, by considering both the preempted and the preempting tasks, would cause the MILP size to explode, as the impact on the cache of all intermediate tasks which execute during the preemption has to be considered.

VI. Prospects

The methods presented before often aim at solving a specific issue related to the use of caches in real-time embedded

systems. Some focus on solving the predictability problem by either accounting for the CRPD in the WCET or during the scheduling analysis, or by eliminating those delays using for example a fully-partitioned cache or static locking. On the other hand, some authors prefer to focus on the schedulability problem using for example limited preemption techniques.

Along with many approaches to reduce either the CRPD or the number of preemptions, a cache-aware schedulability analysis is proposed to ensure predictability, such as [25] for cache hybrid-partitioning or [20] for preemption threshold scheduling. An other solution can be to associate cache partitioning to reduce inter-task conflicts and cache locking to reduce the intra-task interference, as proposed in [113].

The two problems of predictability and schedulability are hard to solve at the same time. For example, fully-partitioning ensure predictability as no CRPD is incurred. However, the consequent increase in task WCETs may threaten the system schedulability. To overcome this issue, an interesting solution is to combine different approaches focusing on different goals. The work presented in [114] is a good example of what can be done in this direction. The authors combine partitioning with preemption threshold scheduling. They use the fact that considering preemption thresholds create groups of non-preempting tasks. So these tasks can share a common cache partition without incurring CRPDs. So the advantage of hybrid-partitioning (which allows a trade-off between reducing extrinsic cache interference, i.e. CRPDs, without increasing too much the intrinsic cache interference, and as a result the WCET) is coupled with the benefit of fully-partitioning (which is to eliminate all CRPDs).

An other solution is to apply cache-aware schedulability analyses to determine thresholds [20] or task priorities [112]. Memory layouts can also be used in combination with limited preemption policies. Cache locking could also be used along with FPP scheduling as the number of memory blocks needing to be reloaded (and so which would have to be locked) might be quite smaller (when using an analysis such as in [32]).

VII. Conclusion

This paper has proposed a state of the art of different strategies to deal with cache issues in uniprocessor real-time scheduling. The presented methods aim either at tightening CRPD bounds to improve predictability or modifying existing scheduling policies to reduce these CRPDs in order to increase the system schedulability. What can easily be seen is that no solution clearly outperforms the other ones. Moreover, very often, no optimal solution exists such as proved for cache-aware scheduling or cache locking.

Combination between the different methods can allow significant improvements. For example, the use of cache partitioning/locking or memory layout enables to decrease extrinsic interference which in turn decrease the cost of a preemption. Then using a limited preemptive policy such as FPP reduce the number of preemptions and so the total CRPD. Finally, using a cache-aware schedulability analysis ensures predictability and avoid wasting hardware resources.

Future works could be conducted on improving these combinations. Moreover, it seems interesting to us to study further the problem of optimal cache-aware scheduling in order to find if effective heuristics can be found.

This problem of cache interference is even bigger when dealing with multiprocessors. Indeed, multiprocessors use cache levels with private and shared caches, making the cache analysis problem much more complex, see [125]. Moreover, when considering global scheduling, tasks or jobs can migrate from one core to another one, causing additional delays called cache-related migration delays [48].

References

[1] Federal Aviation Administration. *Commercial Off-The-Shelf Real-Time Operating System and Architectural Considerations*. 2004. URL: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/03-77_COTS_RTOS.pdf.

[2] Y. Allard, G. Nelissen, J. Goossens, and D. Milojevic. "A context aware cache controller to bridge the gap between theory and practice in real-time systems." In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*. 2014, pp. 1–10.

[3] S. Altmeyer and C. Burguiere. "A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay." In: *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*. 2009, pp. 109–118.

[4] S. Altmeyer, C. Burguiére, and R. Wilhelm. "Computing the Maximum Blocking Time for Scheduling with Deferred Preemption." In: *Future Dependable Distributed Systems, 2009 Software Technologies for*. 2009, pp. 200–204.

[5] S. Altmeyer, R.I. Davis, and C. Maiza. "Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems." In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. 2011, pp. 261–271.

[6] S. Altmeyer, R. Douma, W. Lunniss, and R.I. Davis. "OUTSTANDING PAPER: Evaluation of Cache Partitioning for Hard Real-Time Systems." In: *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. 2014, pp. 15–26.

[7] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. *Pre-emption Cost Aware Response Time Analysis for Fixed Priority Pre-emptive Systems*. Tech. rep. University of York, Department of Computer Science, 2011.

[8] Sebastian Altmeyer and Gernot Gebhard. "WCET Analysis for Preemptive Scheduling." In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*. Ed. by Raimund Kirner. Vol. 8. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008.

[9] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. "Resilience Analysis: Tightening the CRPD Bound for Set-associative Caches." In: *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages,*

Compilers, and Tools for Embedded Systems. LCTES '10. Stockholm, Sweden, 2010, pp. 153–162.

[10] A Arnaud and I Puaut. "Dynamic instruction cache locking in hard real-time systems." In: *Proc. of the 14th Int. Conference on Real-Time and Network Systems*. 2006, pp. 179–188.

[11] S. Baruah. "The limited-preemption uniprocessor scheduling of sporadic task systems." In: *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*. 2005, pp. 137–144.

[12] Andrea Bastoni, Björn Brandenburg, and James Anderson. "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability." In: *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*. 2010, pp. 33–44.

[13] Swagato Basumallick and Kelvin Nilsen. "Cache issues in real-time systems." In: *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*. Vol. 5. 1994.

[14] E. Berg and E. Hagersten. "StatCache: a probabilistic approach to efficient and accurate data locality analysis." In: *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*. 2004, pp. 20–27.

[15] M. Bertogna and S. Baruah. "Limited Preemption EDF Scheduling of Sporadic Task Systems." In: *Industrial Informatics, IEEE Transactions on* 6.4 (2010), pp. 579–591.

[16] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. "Optimal Selection of Preemption Points to Minimize Preemption Overhead." In: *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*. 2011, pp. 217–227.

[17] M. Bertogna, G. Buttazzo, M. Marinoni, Gang Yao, F. Esposito, and M. Caccamo. "Preemption Points Placement for Sporadic Task Sets." In: *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*. 2010, pp. 251–260.

[18] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. "Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited." In: *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*. 2007, pp. 269–279.

[19] R.J. Bril, M.M.H.P. van den Heuvel, U. Keskin, and J.J. Lukkien. "Generalized Fixed-Priority Scheduling with Limited Preemptions." In: *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. 2012, pp. 209–220.

[20] R.J. Bril, S. Altmeyer, M.M.H.P. Van Heuvel, R.I. Davis, and M. Behnam. "Integrating Cache-Related Pre-Emption Delays into Analysis of Fixed Priority Scheduling with Pre-Emption Thresholds." In: *Real-Time Systems Symposium (RTSS), 2014 IEEE*. 2014, pp. 161–172.

[21] B.D. Bui, M. Caccamo, Lui Sha, and J. Martinez. "Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems." In: *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*. 2008, pp. 101–110.

[22] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. "Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions." In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Ed. by Niklas Holsti. Vol. 10. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 1–11.

[23] Claire Burguière and Christine Rochange. "On the Complexity of Modeling Dynamic Branch Predictors when Computing Worst-Case Execution Time." In: *Proceedings of the ERCIM/DECOS Workshop On Dependable Embedded Systems*. 2007.

[24] J.V. Busquets-Mataix, J.J. Serrano-Martin, R. Ors-Carot, P. Gil, and A. Wellings. "Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems." In: *Real-Time Systems, 1996., Proceedings of the Eighth Euromicro Workshop on*. 1996, pp. 271–276.

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

[26] G.C. Buttazzo, M. Bertogna, and Gang Yao. "Limited Preemptive Scheduling for Real-Time Systems. A Survey." In: *Industrial Informatics, IEEE Transactions on* 9.1 (2013), pp. 3–15.

[27] Giorgio C. Buttazzo. *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*. 3rd ed. Vol. 24. Real-Time System Series. Springer US, 2011.

[28] "Cache-related preemption delay via useful cache blocks: Survey and redefinition." In: *Journal of Systems Architecture* 57.7 (2011). Special Issue on Worst-Case Execution-Time Analysis, pp. 707 –719.

[29] J.M. Calandrino and J.H. Anderson. "Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study." In: *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*. 2008, pp. 299–308.

[30] A.M. Campoy, A. Perles, F. Rodriguez, and J.V. Busquets-Mataix. "Static use of locking caches vs. dynamic use of locking caches for real-time systems." In: *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*. Vol. 2. 2003, 1283–1286 vol.2.

[31] Marti Campoy, A Perles Ivars, and JV Busquets-Mataix. "Static use of locking caches in multitask preemptive real-time systems." In: *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*. 2001, pp. 1–6.

[32] J. Cavicchio, C. Tessler, and N. Fisher. “Minimizing Cache Overhead via Loaded Cache Blocks and Preemption Placement.” In: *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. 2015, pp. 163–173.

[33] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoit Triquet, and Reinhard Wilhelm. “Predictability Considerations in the Design of Multi-Core Embedded Systems.” In: *Proceedings of Embedded Real Time Software and Systems*. 2010, pp. 36–42.

[34] Peter J. Denning. “The Working Set Model for Program Behavior.” In: *Commun. ACM* 11.5 (May 1968), pp. 323–333.

[35] Huping Ding, Yun Liang, and Tulika Mitra. “Integrated Instruction Cache Analysis and Locking in Multitasking Real-time Systems.” In: *Proceedings of the 50th Annual Design Automation Conference*. DAC ’13. Austin, Texas, 2013, 147:1–147:10.

[36] Huping Ding, Yun Liang, and Tulika Mitra. “WCET-centric Dynamic Instruction Cache Locking.” In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE ’14. Dresden, Germany, 2014, 27:1–27:6.

[37] R. Dobrin and G. Fohler. “Reducing the number of preemptions in fixed priority scheduling.” In: *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*. 2004, pp. 144–152.

[38] Arvind Easwaran, Insik Shin, Insup Lee, and Oleg Sokolsky. *Bouding Preemptions under EDF and RM Schedulers*. Tech. rep. Department of Computer and Information Science, University of Pennsylvania, 2006.

[39] “Efficient and Precise Cache Behavior Prediction for Real-Time Systems.” In: *Real-Time Systems* 17.2-3 (1999).

[40] Heiko Falk and Helena Kotthaus. “WCET-driven Cache-aware Code Positioning.” In: *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES ’11. Taipei, Taiwan, 2011, pp. 145–154.

[41] Heiko Falk, Sascha Plazar, and Henrik Theiling. “Compile-time Decided Instruction Cache Locking Using Worst-case Execution Paths.” In: *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’07. Salzburg, Austria, 2007, pp. 143–148.

[42] “Fast and Precise WCET Prediction by Separated Cache and Path Analyses.” In: *Real-Time Systems* 18.2-3 (2000).

[43] “Feasibility analysis under fixed priority scheduling with limited preemptions.” In: *Real-Time Systems* 47.3 (2011).

[44] “The olympus attitude and orbital control system: A case study in hard real-time system design and implementation.” In: *Ada - Europe ’93*. Ed. by Michel Gauthier. Vol. 688. Lecture Notes in Computer Science. 1993.

[45] Gernot Gebhard and Sebastian Altmeyer. “Optimal Task Placement to Improve Cache Performance.” In: *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*. EMSOFT ’07. Salzburg, Austria, 2007, pp. 259–268.

[46] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. “Cache-aware Scheduling and Analysis for Multicores.” In: *Proceedings of the Seventh ACM International Conference on Embedded Software*. EMSOFT ’09. Grenoble, France, 2009, pp. 245–254.

[47] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. “The Mälardalen WCET Benchmarks – Past, Present and Future.” In: ed. by Björn Lisper. Brussels, Belgium: OCG, July 2010, pp. 137–147.

[48] Damien Hardy and Isabelle Puaut. “Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches.” In:

[49] C.A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. “Bounding pipeline and instruction cache performance.” In: *Computers, IEEE Transactions on* 48.1 (1999), pp. 53–70.

[50] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[51] Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien. “Using fixed-priority scheduling with deferred preemption to exploit fluctuating network bandwidth.” In: *Proceedings Work in Progress (WiP) Session of the 20th Euromicro Conference on Real-Time Systems (ECRTS’08)*. 2008, pp. 40–43.

[52] “Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems.” In: *Real-Time Systems* 48.5 (2012).

[53] “Instruction cache locking for multi-task real-time embedded systems.” In: *Real-Time Systems* 48.2 (2012).

[54] M. Joseph and P. Pandya. “Finding Response Times in a Real-Time System.” In: 29.5 (1986), pp. 390–395.

[55] U. Keskin, R.J. Bril, and J.J. Lukkien. “Exact response-time analysis for fixed-priority preemption-threshold scheduling.” In: *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*. 2010, pp. 1–4.

[56] D.B. Kirk. “SMART (strategic memory allocation for real-time) cache design.” In: *Real Time Systems Symposium, 1989., Proceedings.* 1989, pp. 229–237.

[57] D.B. Kirk and J.K. Strosnider. “SMART (strategic memory allocation for real-time) cache design using the MIPS R3000.” In: *Real-Time Systems Symposium, 1990. Proceedings., 11th*. 1990, pp. 322–330.

[58] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. “Analysis of cache-related preemption delay in fixed-priority preemptive scheduling.” In: *Computers, IEEE Transactions on* 47.6 (1998), pp. 700–713.

[59] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. “Enhanced

analysis of cache-related preemption delay in fixed-priority preemptive scheduling." In: *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*. 1997, pp. 187–198.

[60] Jinkyu Lee and K.G. Shin. "Preempt a Job or Not in EDF Scheduling of Uniprocessor Systems." In: *Computers, IEEE Transactions on* 63.5 (2014), pp. 1197–1206.

[61] Yau-Tsun Steven Li and Sharad Malik. "Performance Analysis of Embedded Software Using Implicit Path Enumeration." In: *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, &Amp; Tools for Real-time Systems*. LCTES '95. La Jolla, California, USA, 1995, pp. 88–98.

[62] J. Liedtke, H. Hartig, and M. Hohmuth. "OS-controlled cache predictability for real-time systems." In: *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*. 1997, pp. 213–224.

[63] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61.

[64] Tiantian Liu, Minming Li, and C.J. Xue. "Minimizing WCET for Real-Time Embedded Systems via Static Instruction Cache Locking." In: *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*. 2009, pp. 35–44.

[65] P. Lokuciejewski, H. Falk, and P. Marwedel. "WCET-driven Cache-based Procedure Positioning Optimizations." In: *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*. 2008, pp. 321–330.

[66] T. Lundqvist and P. Stenstrom. "Timing anomalies in dynamically scheduled microprocessors." In: *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*. 1999, pp. 12–21.

[67] W. Lunniss, S. Altmeyer, C. Maiza, and R.I. Davis. "Integrating cache related pre-emption delay analysis into EDF scheduling." In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. 2013, pp. 75–84.

[68] Will Lunniss, Sebastian Altmeyer, and Robert Davis. "A Comparison between Fixed Priority and EDF Scheduling accounting for Cache Related Pre-emption Delays." In: *Leibniz Transactions on Embedded Systems* 1.1 (2014).

[69] Will Lunniss, Sebastian Altmeyer, and Robert I. Davis. "Optimising Task Layout to Increase Schedulability via Reduced Cache Related Pre-emption Delays." In: *Proceedings of the 20th International Conference on Real-Time and Network Systems*. RTNS '12. Pont à Mousson, France, 2012, pp. 161–170.

[70] Will Lunniss, Sebastian Altmeyer, Giuseppe Lipari, and Robert I. Davis. "Accounting for Cache Related Pre-emption Delays in Hierarchical Scheduling." In: *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. RTNS '14. Versaille, France, 2014, 183:183–183:192.

[71] J. Marinho, V. Nelis, and S.M. Petters. "Temporal isolation with preemption delay accounting." In: *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. 2014, pp. 1–8.

[72] J.M. Marinho, V. Nelis, S.M. Petters, and I. Puaut. "An improved preemption delay upper bound for floating non-preemptive region." In: *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*. 2012, pp. 57–66.

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

[74] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. "Evaluation Techniques for Storage Hierarchies." In: *IBM Syst. J.* 9.2 (June 1970), pp. 78–117.

[75] "Measuring the Performance of Schedulability Tests." In: *Real-Time Systems* 30.1-2 (2005).

[76] "An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms." In: *Algorithms for Memory Hierarchies*. Ed. by Ulrich Meyer, Peter Sanders, and Jop Sibeyn. Vol. 2625. Lecture Notes in Computer Science. 2003.

[77] E. Mezzetti and T. Vardanega. "A rapid cache-aware procedure positioning optimization to favor incremental development." In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. 2013, pp. 107–116.

[78] Jeffrey C. Mogul and Anita Borg. "The Effect of Context Switches on Cache Performance." In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA, 1991, pp. 75–84.

[79] Frank Mueller. "Compiler Support for Software-based Cache Partitioning." In: *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, &Amp; Tools for Real-time Systems*. LCTES '95. La Jolla, California, USA, 1995, pp. 125–133.

[80] Hemendra Singh Negi, Tulika Mitra, and Abhik 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*. CODES+ISSS '03. Newport Beach, CA, USA, 2003, pp. 201–206.

[81] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. "PapaBench: a Free Real-Time Benchmark." In: *WCET* 4 (2006).

[82] R. Pellizzoni and M. Caccamo. "Toward the Predictable Integration of Real-Time COTS Based Systems." In: *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. 2007, pp. 73–82.

[83] Bo Peng, N. Fisher, and M. Bertogna. "Explicit Preemption Placement for Real-Time Conditional Code." In: *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. 2014, pp. 177–188.

[84] S.M. Petters. "Bounding the execution time of real-time tasks on modern processors." In: *Real-Time Com-*

puting Systems and Applications, 2000. Proceedings. Seventh International Conference on. 2000, pp. 498–502.

[85] Guillaume Phavorin, Pascal Richard, and Claire Maiza. "Complexity of scheduling real-time tasks subjected to cache-related preemption delays." In: *Emerging Technologies and Factory Automation (ETFA), 2015 IEEE Conference on*. 2015, pp. 1–8.

[86] Guillaume Phavorin, Pascal Richard, and Claire Maiza. *Static CRPD-Aware Real-Time Scheduling*. Work-in-Progress session of the 27th Euromicro Conference on Real-Time Systems (ECRTS'2015). 2015.

[87] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. "WCET-aware software based cache partitioning for multi-task real-time systems." In: *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*. 2009, pp. 78–88.

[88] I. Puaut. "WCET-centric software-controlled instruction caches for hard real-time systems." In: *Real-Time Systems, 2006. 18th Euromicro Conference on*. 2006, 10 pp.–226.

[89] I. Puaut and C. Pais. "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison." In: *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*. 2007, pp. 1–6.

[90] Isabelle Puaut. *Architecture des processeurs et vérification de contraintes de temps-réel strict*. 2002.

[91] Isabelle Puaut and Christophe Pais. *Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison*. Tech. rep.

[92] H. Ramaprasad and F. Mueller. "Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks." In: *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*. 2006, pp. 71–80.

[93] H. Ramaprasad and F. Mueller. "Bounding Worst-Case Response Time for Tasks with Non-Preemptive Regions." In: *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*. 2008, pp. 58–67.

[94] H. Ramaprasad and F. Mueller. "Tightening the Bounds on Feasible Preemption Points." In: *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*. 2006, pp. 212–224.

[95] J. Reineke, S. Altmeyer, D. Grund, S. Hahn, and C. Maiza. "Selfish-LRU: Preemption-aware caching for predictability and performance." In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. 2014, pp. 135–144.

[96] Jan Reineke. "Caches in WCET Analysis." PhD thesis. Univeristät des Saarlandes, 2008.

[97] Jan Reineke and Daniel Grund. "Relative Competitive Analysis of Cache Replacement Policies." In: *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '08. Tucson, AZ, USA, 2008, pp. 51–60.

[98] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. "A Definition and Classification of Timing Anomalies." In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Ed. by Frank Mueller. Vol. 4. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006.

[99] "ILP-Based Interprocedural Path Analysis." In: *Embedded Software*. Ed. by Alberto Sangiovanni-Vincentelli and Joseph Sifakis. Vol. 2491. Lecture Notes in Computer Science. 2002.

[100] "Scalable and precise refinement of cache timing analysis via path-sensitive verification." In: *Real-Time Systems* 49.4 (2013).

[101] "Integrated Intra- and Inter-task Cache Analysis for Preemptive Multi-tasking Real-Time Systems." In: *Software and Compilers for Embedded Systems*. Ed. by Henk Schepers. Vol. 3199. Lecture Notes in Computer Science. 2004.

[102] J. Schneider. "Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems." In: *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*. 2000, pp. 195–204.

[103] J. Simonson and J.H. Patel. "Use of preferred preemption points in cache-based real-time systems." In: *Computer Performance and Dependability Symposium, 1995. Proceedings., International*. 1995, pp. 316–325.

[104] Jan Staschulat and Rolf Ernst. "Scalable Precision Cache Analysis for Real-time Software." In: *ACM Trans. Embed. Comput. Syst.* 6.4 (Sept. 2007).

[105] Yudong Tan and Vincent Mooney. "Timing Analysis for Preemptive Multitasking Real-time Systems with Caches." In: *ACM Trans. Embed. Comput. Syst.* 6.1 (Feb. 2007).

[106] Yudong Tan and Vincent J. Mooney III. "WCRT Analysis for a Uniprocessor with a Unified Prioritized Cache." In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '05. Chicago, Illinois, USA, 2005, pp. 175–182.

[107] "Techniques to increase the schedulable utilization of cache-based preemptive real-time systems1." In: *Journal of Systems Architecture* 46.4 (2000), pp. 357 –378.

[108] Stephan Thesing. "Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models." eng. PhD thesis. Postfach 151141, 66041 Saarbrcken: Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes, 2004.

[109] "Timing Analysis for Instruction Caches." In: *Real-Time Systems* 18.2-3 (2000).

[110] Hiroyuki Tomiyama and Nikil 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*. CODES '00. San Diego, California, USA, 2000, pp. 67–71.

[111] Hiroyuki Tomiyama and Hiroto Yasuura. “Code Placement Techniques for Cache Miss Rate Reduction.” In: *ACM Trans. Des. Autom. Electron. Syst.* 2.4 (Oct. 1997), pp. 410–429.

[112] Hai-Nam Tran, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. “Addressing cache related preemption delay in fixed priority assignment.” In: *Emerging Technologies and Factory Automation (ETFA), 2015 IEEE Conference on*. 2015, pp. 1–8.

[113] X. Vera, B. Lisper, and Jingling Xue. “Data caches in multitasking hard real-time systems.” In: *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*. 2003, pp. 154–165.

[114] Chao Wang, Zonghua Gu, and Haibo Zeng. “Integration of Cache Partitioning and Preemption Threshold Scheduling to Improve Schedulability of Hard Real-Time Systems.” In: *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. 2015, pp. 69–79.

[115] Yun Wang and M. Saksena. “Scheduling fixed-priority tasks with preemption threshold.” In: *Real-Time Computing Systems and Applications, 1999. RTCSA ’99. Sixth International Conference on*. 1999, pp. 328–335.

[116] Bryan C. Ward, Abhilash Thekkilakattil, and James H. Anderson. “Optimizing Preemption-Overhead Accounting in Multiprocessor Real-Time Systems.” In: *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. RTNS ’14. Versaille, France, 2014, 235:235–235:243.

[117] S. Wasly and R. Pellizzoni. “Hiding memory latency using fixed priority scheduling.” In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. 2014, pp. 75–86.

[118] R.T. White, F. Mueller, C.A. Healy, D.B. Whalley, and M.G. Harmon. “Timing analysis for data caches and set-associative caches.” In: *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*. 1997, pp. 192–202.

[119] Jack Whitham, Neil C. Audsley, and Robert I. Davis. “Explicit Reservation of Cache Memory in a Predictable, Preemptive Multitasking Real-time System.” In: *ACM Trans. Embed. Comput. Syst.* 13.4s (Apr. 2014), 120:1–120:25.

[120] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. “The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools.” In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53.

[121] Andrew Wolfe. “Software-based Cache Partitioning for Real-time Applications.” In: *J. Comput. Softw. Eng.* 2.3 (Mar. 1994), pp. 315–327.

[122] “Worst Case Execution Time Analysis for a Processor with Branch Prediction.” In: *Real-Time Systems* 18.2-3 (2000).

[123] “Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption.” In: *Real-Time Systems* 42.1-3 (2009).

[124] Gang Yao, G. Buttazzo, and M. Bertogna. “Feasibility Analysis under Fixed Priority Scheduling with Fixed Preemption Points.” In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*. 2010, pp. 71–80.

[125] Wei Zhang and Jun Yan. “Accurately Estimating Worst-Case Execution Time for Multi-core Processors with Shared Direct-Mapped Instruction Caches.” In: *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA ’09. 15th IEEE International Conference on*. 2009, pp. 455–463.