Bounding the Effects of Resource Access Protocols on Cache Behavior

Enrico Mezzetti[1], Marco Panunzio[1], and Tullio Vardanega[1]

1 University of Padua, Department of Pure and Applied Mathematics, via Trieste, 63 35121 Padua, Italy
`{emezzett,panunzio,tullio.vardanega}@math.unipd.it`

Abstract

The assumption of task independence has long been consubstantial with the formulation of many schedulability analysis techniques. That assumption is evidently advantageous for the mathematical formulation of the analysis equations, but ill fit to capture the actual behavior of the system. Resource sharing is one of the system design dimensions that break the assumption of task independence. By shaking the very foundations of the real-time analysis theory, the advent of multicore systems has caused resurgence of interest in resource sharing and synchronization protocols, and also dawned the fact that the assumption of task independence may be forever broken. Research in cache-aware schedulability analysis instead has paid very little attention to the impact that synchronization protocols may have on cache behavior. A blocked task may in fact incur time penalties similar in kind to those caused by preemption, in that some useful code or data already loaded in the cache may be evicted while the task is blocked. In this paper we characterize the sources of cache-related blocking delay (CRBD). We then provide a bound on the CRBD for three synchronization protocols of interest. The comparison between these bounds provides striking evidence that an informed choice of the synchronization protocol helps contain the perturbing effects of blocking on the cache state.

Keywords and phrases Resource access protocols, cache, worst-case response time

Digital Object Identifier 10.4230/OASIcs.WCET.2010.11

1 Introduction

The correctness of schedulability analysis techniques for preemptive real-time systems relies on the use of safe estimates of both the worst-case execution time (WCET) of the system tasks and the additional costs due to interrupts and task preemptions. Whereas the determination of safe and tight WCET bounds is a widely acknowledged and studied problem [14], most schedulability analysis techniques rest on the simplifying assumption of constant (and negligible) context-switch costs. Unfortunately, the use of hardware acceleration features like caches and complex pipelines breaks this assumption for good. With the adoption of caches, in particular, the context-switch cost is no longer constant as it must account for the interferences between tasks: interrupt handling and preemption may influence the execution time of a preempted task. On resumption in fact, the preempted task may incur a number of additional cache misses as some useful cache contents may have been evicted from the cache. Cache-aware schedulability analysis techniques [5, 12, 13] aim at including those cache effects in the schedulability analysis by accounting for the so-called cache-related preemption delay (CRPD) overheads in the response time of individual tasks.

However, interference caused by tasks is not limited to task preemptions. The assumption of task independence rarely holds in practice and real systems often include shared resources which multiple tasks can access in mutual exclusion. Task blocking therefore occurs, which causes priority inversion to arise, and the need for resource access protocols to bound it. In response time analysis (RTA) for fixed-priority systems, the time a task is forced to wait for a shared resource in use by lower priority tasks (the task *blocking time*) is assumed to be bounded.

When it comes to cache interference, however, task blocking may cause effects similar in kind to task preemption, in that some useful code or data blocks already loaded in the cache may be evicted while the task is being blocked. Very few works [10] consider the additional time spent in reloading the evicted cache contents, which is referred to as Cache-Related Blocking Delay (CRBD). Although a task typically suffers from blocking as it shares some resources with lower priority tasks, different patterns and durations of blocking – and thus the amplitude of the CRBD – can be induced by the specific resource access policy in use.

We contend that the CRBD cannot be dismissed as a negligible cache-related effect, and should instead be accounted for by cache-aware schedulability analysis. The contribution of this paper is a characterization of the effects of blocking on the cache behavior (i.e., CRBD) in fixed-priority preemptive systems and the formalization of a worst-case bound of the incurred delay under different resource access protocols. The rest of the paper is organized as follows: section 2 surveys the state-of-the art approaches to cope with cache interference between tasks; section 3 provides a formal definition of the CRBD and defines a bound to it for three resource access protocols of interest; section 4 finally draws some conclusions.

2 Related Work

In general, WCET analysis approaches focus on intra-task cache behavior and, while not directly accounting for inter-task (i.e., extrinsic) interference, try to include the cache effects of the latter in schedulability analysis. In this paper we do not consider limited-preemptible systems, cache partitioning or locking techniques, which may attenuate inter-task interference.

Since task preemption is typically regarded as the main source of interference between tasks, the inclusion of cache effects in schedulability analysis is generally accomplished by accounting for an upper bound on the CRPD in the response time of individual tasks. For example, the RTA iterative equation for task τ_i has been extended to include cache effects due to preemptions as follows:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times (C_j + \gamma_j) \tag{1}$$

where w_i^k refers to the time window under analysis, C_i and B_i are respectively the WCET and maximum blocking time for the analyzed task, and the remaining term represents the *interference* from higher priority tasks, which includes the induced CRPD γ_j [5].

The exact CRPD depends on both the preempted and the preempting tasks since it captures the time required by the preempted task to reload cache blocks that have been evicted by the preempting tasks and that will be reused when the preempted task resumes execution.

Two basic concepts are useful to understand the bounds on cache interference between tasks:

- *Useful Cache Blocks* (UCB): cache blocks that may be referenced again before they could be evicted by other memory blocks, according to the cache replacement policy;
- *Used Cache Blocks* ($\overline{UCB}$): cache blocks that may be accessed during the execution of the preempting task.

Earlier approaches [5, 6] that relied on UCB or $\overline{UCB}$ alone to compute the CRPD bound were overly conservative as neither all UCB would be always evicted nor the $\overline{UCB}$ would necessarily evict useful blocks. Less pessimistic approaches have been suggested in [7, 13, 12], which account for both UCB and $\overline{UCB}$ to compute the CRPD bound. Moreover, acknowledging the fact that a cache block may not be useful (or used) along every program path, Negi et al. [9] introduce the more elaborate notions of Cache Utility Vector (CUV) and Final Usage Vector (FUV) to capture all possible cache states along the different execution paths of both the preempted and the preempting tasks. A refinement in UCB computation, combined with WCET analysis, has been proposed in [1]. More recently, with respect to LRU set-associative caches, the idea of *resilience* has been introduced [2] to exclude from the CRPD computation those UCB that can be guaranteed to persist in the cache, thanks to the specific replacement policy.

The term B_i in Equation 1 refers to an upper bound to the blocking time suffered by task τ_i due to resource contention. As observed in [10], similarly to the interference from higher priority tasks, the interference from lower priority tasks contending for shared resources should be considered to predict cache behavior. The effects of blocking are similar to those related to the CRPD since a lower priority task may evict some useful cache blocks of a higher priority task, which thus incurs some CRBD.

The work in [10] extends previous work by the same authors on fully-preemptive and non-preemptive task regions, to cope with shared resources under the Priority Inheritance Protocol. They use a complex framework that exploits task phasing to account for both CRPD and CRBD in the response time of a task. The computation of the CRBD (limited to data cache) employs the same concepts as used to compute the CRPD. In contrast to [10], in this paper we focus on computing an upper bound on the CRBD under different protocols, rather than on its inclusion in the worst-case response time analysis.

3 Cache-Related Blocking Delay

Shared resources typically need to be accessed in mutual exclusion. When a high priority task needs to access a resource that is already locked by a lower priority task, it cannot proceed until the lower priority task completes execution inside the resource and relinquishes its lock. Whenever a lower priority task prevents the execution of a higher priority task, the system experiences potentially unbounded priority inversion. This phenomenon can be bounded with the use of a resource access protocol. In this paper we focus on three well-known protocols: the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP) and the Immediate Ceiling Protocol (ICP).

In a fixed-priority preemptive system with shared resources equipped with a synchronization protocol, three different kinds of blocking may arise [8]:

- *Direct blocking* occurs when a higher priority task requests a shared resource held by a lower priority task; another form of direct blocking, *transitive blocking*[1], occurs when nested resources access is permitted, and a blocked task transitively suffers from the blocking incurred by the blocking task itself.
- *Inheritance or push-through blocking* occurs for a task τ_m that does not need any shared resource, when a lower priority task τ_j blocks a task τ_i with priority $\pi(\tau_i) > \pi(\tau_m)$ and executes at a priority higher than $\pi(\tau_m)$ due to some priority inheritance rule.

[1] Also referred to as *chain blocking*.

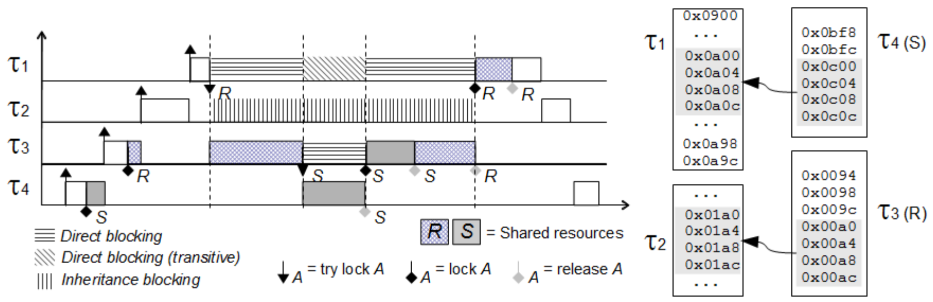


Figure 1 Example of CRBD.

- *Avoidance blocking* occurs when a task τ_i is denied access to an *available* resource to prevent deadlock.

From the standpoint of caches, the execution of the blocking task inside a critical section may cause the eviction of useful cache contents that would have later been reused by the blocked task. A high priority task will thus incur a time penalty (or blocking delay) because of the additional cache misses, regardless of the type of suffered blocking.

The scenario depicted in Figure 1 illustrates the different types of blocking under the PIP and shows how lower priority tasks may affect the cache content of higher priority tasks. In particular, task τ_1 and τ_3 suffer direct blocking, while task τ_2 suffers inheritance blocking. Assume that τ_1 has loaded four cache blocks that would be shortly reused (i.e., the shaded memory addresses in Figure 1) in a small direct mapped instruction cache. Unfortunately, task τ_1 is blocked when trying to access shared resource R currently held by τ_3, which in turns is blocked by τ_4 on the shared resource S. While τ_3 has no effect on the useful cache content of τ_1, the code executed by τ_4 in its critical section accessing S maps exactly to the same cache sets and evicts all the four useful blocks of τ_1. When τ_1 resumes, it will incur a CRBD of four additional cache misses.

A subtler penalty is experienced by task τ_2 due to the execution of τ_3: whereas it does not share any resource with other tasks, τ_2 is blocked due to priority inheritance. In the example, the useful cache content of τ_2 is evicted during the execution of τ_3 inside its critical region. It is worth noting that τ_2 would not have suffered any interference (CRPD) due to the higher priority task τ_1.

Hence, blocking does not only affect the response time of a task as a single worst-case factor independent of the task itself, but it may also directly affect its execution time. This is so because priority inversion, even if bounded, causes a cache-related delay akin to that caused by preemption. Similarly to CRPD, the amount of delay potentially incurred by a task on a single blocking event depends on both the cache content of the blocked task and the execution of the blocking task.

The actual CRBD is thus a function of the UCB (cf. Section 2) of the blocked tasks and the $\overline{UCB}$ of the blocking task. In contrast to CRPD, however, the $\overline{UCB}$ are not determined by the whole execution of the blocking task since the induced delay can stem just from the execution inside critical sections. Furthermore, in case of direct (and some forms of avoidance) blocking, also the UCB of the blocked task are limited to those determined at the beginning of the critical section at which the task is blocked.

Although we can expect the CRBD to be small for single critical sections, its relevance increases as soon as a task may experience several and potentially different blocking events during the same activation. The CRBD arising from such blocking events cannot be disregarded during schedulability analysis as its cumulative effect may invalidate the analysis results.

We performed some initial experiments to gage by static analysis the impact of the CRBD on the instruction cache performance of blocked tasks. We implemented a small test case made up of three tasks: τ_1 and τ_3, reading and/or writing a shared resource, and a notionally independent task τ_2, taken from the Mälardalen benchmark[2]. We extended the Heptane tool from IRISA/Rennes to compute the number of additional cache misses incurred by τ_1 and τ_2 owing to direct blocking and indirect blocking respectively. With a 4 KB, 32 B lines, direct-mapped instruction cache, under PIP, task τ_1, whose stand-alone cache performance shows 30 misses over 1011 cache accesses, may suffer 8 additional cache misses from the CRPD directly induced by τ_3. Task τ_2 may incur 3 further cache misses against a stand-alone cache performance of 9 misses over 54 accesses, when the impact of inheritance blocking is not considered[3].

The CRBD potentially suffered from a task depends on the actual resource access protocol in use, as it determines both the possible types of blocking incurred and the maximum number of blocking events suffered for each activation.

In principle, the CRBD problem could be transformed into a CRPD problem by modeling the critical sections as tasks that may preempt higher priority tasks, thus exploiting the intrinsic similarities between the two phenomena. However, some specialization should still be needed to capture the specificity of blocking as well as of the resource access protocol in force, thus boosting the complexity of the analysis approach considerably. For example, preemption points should be predefined to permit a sound computation of UCB in case of direct blocking. Similarly, one may need to account for the transitivity of blocking depending on the resource access protocols in use. In our approach, instead, we solely focus on the computation of UCB and $\overline{UCB}$ and exploit sound theoretical bounds [11] on the number of blocking events to provide an upper bound on the CRBD.

In the following we first provide a formal characterization of the CRBD potentially incurred by a task; and then we exploit well-known bounds on the number of blocking events suffered by a task, under different resource access protocols to observe that the way in which the worst-case CRBD can affect the execution time of tasks is highly related to the choice of protocol.

3.1 Bounding the Cache-Related Blocking Delay

We assume total ordering between tasks such that $i < j$ if $\pi(\tau_i) > \pi(\tau_j)$: hence τ_0 is the highest priority task. In our model a task τ_i self-suspends only at the end of every execution of its jobs, and may access a shared resource $R \in SR_i$, where SR_i identifies the subset of the system resources (SR) that get accessed by τ_i.

It is worth noting that from a finer-grained point of view, acquiring (respectively releasing) a shared resource corresponds to entering (exiting) a critical section where the resource is locked (unlocked). Since a task τ_i may access a shared resource R through different critical sections, we define cs_i^R to be the set of critical sections in τ_i accessing the resource $R \in SR_i$. Similarly, $cs_{i,k}^R$ identifies the k^{th} critical section in τ_i accessing the resource $R \in SR_i$. In any case, we assume critical sections to be properly nested so that they can never overlap. For every pair of critical sections $cs_{i,k}$, $cs_{i,z}$ in τ_i either $cs_{i,k} \subset cs_{i,z}$,

[2] *http://www.mrtc.mdh.se/projects/wcet/benchmarks.html*
[3] Furthermore, in this case, no additional misses originate from preemption by τ_1.

$cs_{i,k} \supset cs_{i,z}$ or $cs_{i,k} \cap cs_{i,z} = \emptyset$.

The determination of the CRBD incurred by a task exploits similar concepts as when computing the CRPD, involving the computation of UCB and $\overline{UCB}$ for blocked and blocking task respectively. First, we recall that the set of UCB and $\overline{UCB}$ for a task τ_i are dependent on each specific node n in the Control-Flow Graph (CFG) of τ_i, where each node represents a basic block. In fact, UCB and $\overline{UCB}$ can be safely computed at basic block level, as proved in [6].

According to [7, 13, 12] the UCB_i^n for a task τ_i at node n can be computed as the intersection between the sets of *ReachingBlocks* (RB) and *LiveBlocks* (LB) at node n where RB is the set of cache blocks potentially cached at node N, whereas LB is the set of blocks that could potentially be reused in the successors of n. Intuitively, instead, $\overline{UCB}_i^n$ can be computed as $RB_i(n)$. Thus, $UCB_i^n = RB_i(n) \bigcap LB_i(n)$ and $\overline{UCB}_i^n = RB_i(n)$.

In case of blocking, we are interested in determining UCB and $\overline{UCB}$ for a task τ_i blocked on a critical section $cs_{i,k}^R$. For example, let us consider a simple case of direct blocking between two tasks. Task τ_i is blocked when trying to access critical section $cs_{i,k}^R$ because a lower-priority task τ_j is executing inside a critical section $cs \in cs_j^R$ accessing the same shared resource R. In this case, the set of UCB for the blocked task τ_i is to be computed with respect to the node n_R trying to enter $cs_{i,k}^R$.

$$UCB_{i,k}^R = RB_i(n_R) \cap LB_i(n_R), \text{ where } n_R \text{ is the entry node of } cs_{i,k}^R$$

The set of $\overline{UCB}$ for the blocking task τ_j must be computed with respect to the critical section $cs_{j,h}^R$ it is executing within, as only the RBs in $cs_{j,h}^R$ can affect the cache state of τ_i. For this reason, we extend the notion of RB to address *intervals* of nodes in the CFG instead of single nodes.

Given an interval $[n_1, n_2] = \mathcal{I} \in CFG(\tau_i)$, we define $RB_i(\mathcal{I})$ as the contribution to $RB(n_2)$ of all possible paths in $CGF(\tau_i)$ from node n_1 to n_2. Accordingly,

$$\overline{UCB}_j(cs_{j,h}^R) = RB_j(cs_{j,h}^R) = RB_j([first\_node, last\_node]_{cs_{j,h}^R})$$

In the example, the execution of τ_j inside $cs_{j,h}^R$ may evict some useful cache blocks that τ_i may have loaded in the cache before its attempt to enter $cs_{i,k}^R$. The incurred CRBD can be computed as a function of the $UCB_{i,k}^R$ and $\overline{UCB}_{j,h}^R$ terms just defined:

$$CRBD = \otimes_\sigma\big(UCB_{i,k}^R, \overline{UCB}_j(cs_{j,h}^R)\big) \times \text{miss penalty} \tag{2}$$

where the $\otimes_\sigma$ operator accounts for the actual cache associativity and replacement policy in combining the information on useful and used cache blocks, cf. [13, 1]. For example, for direct-mapped caches, $\otimes_{DM}(UCB, \overline{UCB})$ will include those cache sets which at least one cache block in both UCB and $\overline{UCB}$ is mapped to (set intersection). For LRU n-way set-associative caches, instead, the $\otimes_{LRU}$ operator must account for the number of additional cache misses for each cache set. In case of a non-empty $\overline{UCB}$ set, those misses are bounded by the minimum between the cache associativity (n) and the number of UCB mapping to that cache set [4].

In case τ_i and τ_j share more than one resource, we can generalize Equation 2 to determine an upper bound on the delay suffered by τ_i, due to a *single* direct blocking by τ_j for any critical section accessing any shared resource as follows:

$$CRBD_{i,j} \leq \max_{\substack{R \in SR_i, k \in [1, |cs_i^R|] \\ cs \in cs_j^R}} \big\{ \otimes_\sigma \big(UCB_{i,k}^R, \overline{UCB}_j(cs)\big)\big\} \times \text{miss penalty} \tag{3}$$

However, Equation 3 just holds in this simple case where neither transitive direct blocking nor other types of blocking are taken into account. In terms of CRBD, determining the effects of inheritance blocking is much more complex, as the computation of UCB for the blocked task cannot make any simplifying assumption on when the task actually gets blocked.

A more comprehensive bound on the CRBD incurred by a task can be computed by leveraging on the bounds that a specific resource access protocol places on blocking. An upper bound on the worst-case number of blocking events incurred by a task is given in [11, 3] for each protocol. That bound is then combined with the worst-case duration of each critical section to derive a bound on the blocking time potentially suffered by a task. Those bounds typically rely on the notion of potentially blocking critical sections to account for any type of blocking that may occur under the protocol itself. To this end, $\beta_{i,j}$ is defined in [11] as the set of critical sections of a lower-priority task τ_j which can block τ_i in any way. The bounds on the number of blocking events and blocking time exploit the $\beta^*_{i,j}$ set which identifies the set of *outermost* critical sections of τ_j that can block τ_i. More formally:

$$\beta^*_{i,j} = \{(cs_{j,k} | cs_{j,k} \in \beta_{i,j}) \wedge (\neg \exists cs_{j,m} \in \beta_{i,j}, cs_{j,k} \subset cs_{j,m})\}$$

We will exploit the same concepts, with the only difference that we are not interested in the critical section that may incur the maximum blocking time since we focus on the CRBD, which is independent of the duration of the critical section. Instead, we are interested in the critical section $cs_{j,k} \in \beta^*_{i,j}$ which causes the eviction of the greatest number of useful blocks for the blocked task, for all lower-priority tasks τ_j.

In the following, we will combine given bounds on the number of blocking events with the same concepts as used in CRPD analysis to provide a *safe* upper bound on the CRBD under different protocols.

3.2 CRBD under the Priority Inheritance Protocol

When access to shared resources is managed with PIP [11], whenever a task that holds the lock of a resource blocks a higher-priority task, it inherits the priority of the highest-priority task it is blocking[4]. When a task releases the lock of a resource, its priority is lowered to the *highest inherited* priority value[5] [15].

PIP is interesting as it does bound priority inversion and also does not require any knowledge on the system's tasks and their priorities, since the priority value to inherit is determined dynamically. Unfortunately, PIP does not prevent deadlock (which may occur in case of nested critical sections) and a task can be blocked multiple times during a single activation. In fact, a task τ_i can be blocked for the duration of at most $\min(n, m)$ outermost critical sections, where n is the number of lower-priority tasks that may block τ_i and m is the number of semaphores[6] that can be used to block τ_i. In the following we re-elaborate both bounds from the standpoint of the CRBD.

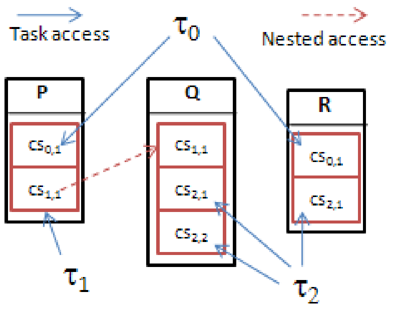
3.2.1 Bound on Lower Priority Tasks

Under PIP, a high priority task τ_H can be blocked by a lower priority task τ_L for at most the duration of one critical section of $\beta^*_{H,L}$. Therefore, given a task τ_i for which there are

[4] This occurs to transitively inherit priority in case of chain blocking.
[5] The original protocol restores the priority to the value inherited before entering the critical section, which is incorrect.
[6] A semaphore corresponds to a shared resource since we assume each resource to be guarded by a binary semaphore.

n lower priority tasks $\{\tau_{i+1}, \dots, \tau_{i+n}\}$, τ_i can be blocked for at most the duration of one critical section in each $\beta^*_{i,k}$, $i+1 \leq k \leq i+n$ [11].

If we assume that all shared resources and critical sections are statically known, we can define a resource access graph and table, similar to that shown in Figure 2.



	P	Q	R
τ_0	$cs^P_{0,1}$		$cs^R_{0,1}$
τ_1	$cs^P_{1,1}$	$cs^Q_{1,1}$	
τ_2		$cs^Q_{2,1}$ $cs^Q_{2,2}$	$cs^R_{2,1}$

Figure 2 Resource graph and corresponding resource access table.

Note that the critical section $cs_{1,1}$ of resource P performs a nested access to critical section $cs_{1,1}$ of resource Q. In this case, the $\beta_{i,j}$ sets derived from Table 2 are as follows: $\beta_{0,1} = \{cs^P_{1,1}, cs^Q_{1,1}\}$ due to resource nesting, $\beta_{1,2} = \{cs^Q_{2,1}, cs^Q_{2,2}, cs^R_{2,1}\}$ (by inheritance blocking), and $\beta_{0,2} = \{cs^R_{2,1}, cs^Q_{2,1}, cs^Q_{2,2}\}$ as τ_2 could transitively block τ_0 by blocking τ_1. The $\beta^*_{i,j}$ sets, instead, removes redundant innermost critical sections; thus, for example, $\beta^*_{0,1} = \{cs^P_{1,1}\}$.

As discussed in Section 3.1, computing the UCB of the blocked task τ_i in case of inheritance blocking needs to consider any possible node in $CFG(\tau_i)$, similarly to task preemption. To avoid the overestimation in considering all possible nodes, we will threat inheritance blocking separately.

An upper bound on the CRBD in case of direct blocking of τ_i due to τ_j is the maximum $\otimes_\sigma$ applied to UCB and $\overline{UCB}$ for any resource accessed by τ_i, every critical section in τ_i accessing that resource and every outermost critical section of τ_j potentially blocking τ_i. Hence, it can be formalized as:

$$CRBD^{base}_{i,j} \leq \max_{\substack{R \in SR_i, k \in [1, |cs^R_i|] \\ cs \in \beta^*_{i,j}}} \left\{ \otimes_\sigma \left(UCB^R_{i,k}, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{4}$$

With regard to inheritance blocking, we need to account for the most penalizing blocking point for τ_i (i.e., node in the CFG). To this end we define $\widehat{\beta}_{i,j}$, a subset of $\beta^*_{i,j}$ including all critical sections in τ_j which can block τ_i due to inheritance blocking. Thus, $\widehat{\beta}_{i,j} = \{cs | cs \in \beta^*_{i,j} \wedge cs$ can block τ_i due to inheritance blocking$\}$. We can now compute the maximum CRBD incurred by τ_i due to inheritance blocking by τ_j as follows:

$$CRBD^{inherit}_{i,j} \leq \max_{\substack{cs \in \widehat{\beta}_{i,j} \\ n \in CFG(\tau_i)}} \left\{ \otimes_\sigma \left(UCB^n_i, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{5}$$

However, since a lower priority task τ_j can block τ_i because it is executing inside at most one $cs \in \beta^*_{i,j}$, each τ_j can induce solely one of either inheritance or "non-inheritance" blocking on τ_i. Hence, we can safely account for the worst-case blocking (inheritance or not), that is:

$$CRBD_i \leq \sum_{j>i} \max \left(CRBD^{base}_{i,j}, CRBD^{inherit}_{i,j} \right) \tag{6}$$

3.2.2 Bound on Semaphores

A second upper bound on blocking, based on the number of semaphores potentially blocking a task under PIP is given in [11]. Under PIP, if there are m semaphores which can block task τ_i, then τ_i can be blocked at most m times, as it can be blocked at most by one critical section for each potentially blocking semaphore. Since we assume that each semaphore corresponds exactly to a shared resource, then τ_i can be blocked at most by one critical section for each potentially blocking resource.

Similarly to the previous case, [11] defines $\xi_{i,j,k}$ as the set of critical sections of a lower-priority task τ_j guarded by a semaphore S_k and which can block τ_i (due to any type of blocking). Subsequently, $\xi^*_{i,j,k}$ identifies the set of all potentially blocking outermost critical sections guarded by S_k, that is $\xi^*_{i,j,k} = \{cs^{S_k}_{j,m} | cs^{S_k}_{j,m} \in \beta^*_{i,j}\}$.

For example, recalling Table 2, $\xi^*_{0,1,P} = \{cs^P_{1,1}\}$, $\xi^*_{0,1,Q} = \{cs^Q_{1,1}\}$, $\xi^*_{0,2,R} = \{cs^R_{2,1}\}$ and $\xi^*_{1,\cdot,R} = \{cs^R_{2,1}\}$ (inheritance). Similarly to the first bound, we define $\widehat{\xi}_{i,j,k}$, a subset of $\xi^*_{i,j,k}$ including all critical sections in $\xi^*_{i,j,k}$ guarded by the semaphore S_k which can block τ_i through inheritance blocking. Thus, $\widehat{\xi}_{i,j,k} = \{cs | cs \in \xi^*_{i,j,k} \wedge cs$ can block τ_i due to inheritance blocking$\}$ can be used to separately account for the direct and inheritance cases. First we provide a means to compute the maximum CRBD for each resource that accounts for any lower priority task and any cs in those tasks that may incur both forms of blocking.

$$CRBD^{base}_{i,R} \leq \max_{\substack{j>i, k\in[1,|cs^R_i|] \\ cs\in\xi^*_{i,j,R}}} \left\{ \otimes_\sigma \left(UCB^R_{i,k}, \overline{UCB}_j(cs)\right)\right\} \times \text{miss penalty} \tag{7}$$

$$CRBD^{inherit}_{i,R} \leq \max_{\substack{n\in CGF(\tau_i) \\ j>i \\ cs\in\widehat{\xi}_{i,j,R}}} \left\{ \otimes_\sigma \left(UCB^n_i, \overline{UCB}_j(cs)\right)\right\} \times \text{miss penalty} \tag{8}$$

Again, since task τ_i can be blocked at most once for each semaphore (resource), we can compute a safe upper bound on the blocking delay by summing the $|S|$ worst-case penalties over the $S \subset SR$ semaphores (resources) potentially blocking τ_i:

$$CRBD_i \leq \sum_{R\in S} \max\left(CRBD^{base}_{i,R}, CRBD^{inherit}_{i,R}\right) \tag{9}$$

The actual bound on the CRBD under PIP is then determined by the minimum between the bounds on lower priority tasks and semaphores (i.e., Equations 6 and 9).

3.3 CRBD under the Priority Ceiling Protocol

With PCP [11], each resource is assigned a *ceiling priority* which is set to at least the priority value of the highest-priority task that uses that resource. Since ceiling priorities are assigned statically, all the tasks of the system and their priority must be known statically. For a task τ_i to be able to access the critical section of a resource, its current priority must be higher than the ceiling priority of any currently locked resource (i.e. semaphore). Otherwise, the task that blocks τ_i inherits the ceiling priority of the resource it is locking.

PCP introduces *avoidance blocking*: a task, when trying to access a resource that is *currently available*, is blocked if its current priority is not higher than the highest ceiling of all semaphores currently locked by other tasks. This protocol rule is used to warrant the absence of deadlock. Furthermore, transitive blocking is not possible, a task τ_i can be blocked at most once per activation, and the duration of the priority inversion is minimized.

Similarly to PIP, a bound on the delay incurred by the effects of blocking on the cache state must account for inheritance blocking separately from direct and avoidance blocking as only the latter ones are triggered when a task attempts to access a resource. Provided that the computation of the $\beta^*_{i,j}$ set includes all critical sections of τ_j that may block[7] τ_i due to direct, inheritance or avoidance blocking, an upper bound for the CRPD can be computed in a similar way to the first bound on PIP. The CRBD suffered by a task τ_i can be bounded by the following equation:

$$CRBD_i \leq \max_{j>i} \left\{ \max \left(CRBD^{base}_{i,j}, CRBD^{inherit}_{i,j} \right) \right\} \tag{10}$$

where $CRBD^{base}_{i,j}$ and $CRBD^{inherit}_{i,j}$ are exactly as defined in the PIP case (Eq. 4 and 5 respectively). As opposed to the PIP case, we are interested just in the most penalizing critical section among all critical sections and all lower-priority tasks, due to Theorem 12 in [11].

3.4 CRBD under the Immediate Ceiling Priority Protocol

The Immediate Ceiling Priority Protocol (ICPP) (direct derivative of Baker's stack resource policy [3]) is similar to PCP, as ceiling priorities are assigned to resources with the same rules. Under ICPP however, a task that enters in a critical section always inherits the ceiling priority, while under PCP only when it is *blocking* a higher-priority task; therefore all tasks with a priority lower than or equal to the ceiling priority cannot be scheduled until the resource has been released. ICPP retains the advantages of PCP: absence of deadlock, tasks can block at most once during each activation and the blocking duration is minimized.

The maximum blocking time for a task τ_i is bounded by the longest outermost critical section executed by a lower-priority task τ_j using a resource with a ceiling priority greater than or equal to the priority of τ_i.

More importantly from the CRBD standpoint, the rules of ICPP prevent any disturbing effects on the cache state of the blocked task. In fact, if blocking occurs, it is always before the affected job begins execution; this implies that cache analysis does not need to account for any effect and can continue to assume the worst-case initial cache state (empty or chaos state, depending on the analysis approach). More formally: $CRBD_i = 0$, $\forall \tau_i$.

3.5 Including the CRBD in Response Time Analysis

A safe bound β_i on the CRBD suffered from each task can be straightforwardly included in the iterative equation of response time analysis. In contrast to CRPD, the delay incurred by blocking does not need to propagate to lower priority tasks since it is separately considered for each task:

$$w^{n+1}_i = C_i + B_i + \beta_i + \sum_{j \in hp(i)} \left\lceil \frac{w^n_i}{T_j} \right\rceil \times (C_j + \gamma_j) \tag{11}$$

where both B_i and β_i depend on the resource access protocol of choice. It is worth noting that the worst-case blocking time and CRBD are not guaranteed to occur altogether. In principle, it could be possible to tighten the computation by accounting for the maximum co-occurrence of B_i and β_i.

[7] Note that the ceiling priority of the resource must be considered when determining potentially blocking critical sections.

As seed for reflection, we note that when it comes to more complex analysis approaches, like e.g., resilience analysis for set-associative caches [2], computing the β_i and γ_j terms separately may *invalidate* the CRPD analysis result, as blocking may incur additional accesses to a cache set.

The comparison of the bounds obtained for the protocols addressed in this paper, though limitedly to their bounds on lower priority tasks[8], shows that ICP is by far preferable with respect to interference on cache as it does not incur any CRBD. The CRBD bounds we provided are pessimistic. Tighter bounds could be computed by straightforwardly extending our approach to a more precise representation for UCB and $\overline{UCB}$ like in [9] or by taking advantage, for example, of task phasing.

4 Conclusion

In this paper we contended that the cache effects caused by the use of synchronization protocols to arbitrate the access to shared resources cannot be dismissed as negligible. Cache contents that are useful to a task of interest may in fact be evicted by lower-priority tasks when the task is blocked. Moreover, different protocols may incur different effects on the task state of the blocked task.

We provided a (pessimistic) bound on the cache-related blocking delay for two well-known protocols: the Priority Inheritance Protocol and the Priority Ceiling Protocol. We also showed that the use of the Immediate Ceiling Protocol does not induce any CRBD, as tasks can be blocked only once per activation and prior to their execution after release.

Although the quantitative effect of the CRBD is not likely to compare with the CRPD, it should not be dismissed as irrelevant: it is arguably important to include it in schedulability analysis that aims to accuracy. We also contend that the cache-related impact should also be contemplated as a distinct evaluation criterion for the selection of the resource access protocol to adopt in a real-time system.

In future work, we plan to define the integer linear problems required for the calculation of the CRBD bounds provided herein, and perform a quantitative estimation of the impact of the CRBD on a representative application case.

Acknowledgements The authors wish to acknowledge the advice of the anonymous reviewers, which helped improve the initial version of this paper, and the support by Damien Hardy at IRISA/Rennes on the Heptane tool.

[8] The bounds that PIP places on semaphores is not straightforwardly comparable with the bound on PCP.

References

1 S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

2 Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: Tightening the crpd bound for set-associative caches. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 153–162. ACM, 2010.

3 T. P. Baker. Stack-based Scheduling for Realtime Processes. *Real-Time Systems*, 3(1):67–99, 1991.

4 Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches - pitfalls and solutions. In *Proc. of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

5 J.V. Busquets-Mataix and A. Wellings. Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems. In *Proc. of the 2nd Real-time Technology and Application Symposium*, 1996.

6 C. Lee, J. Hahn, Y. Seo, S.L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

7 C. G. Lee, K. Lee, J. Hahn, Y. Seo, S. L. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transaction on Software Engineering*, 27(9):805–826, 2001.

8 Jane W. S. Liu. *Real-Time Systems.* Prentice Hall PTR, 2000.

9 Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 201–206. ACM, 2003.

10 H. Ramaprasad and F. Mueller. Bounding worst-case response time for tasks under PIP. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium*, 2009.

11 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

12 J. Staschulat and R. Ernst. Scalable precision cache analysis for preemptive scheduling. In *Proc. of the 2005 ACM SIGPLAN/SIGBED conference on Languages*, 2005.

13 Y. Tan and V. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Proc. of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2004.

14 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G.Bernat, C. Ferdinand, R.Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, G. Staschulat, and P. Stenströem. The worst-case execution time problem: overview of methods and survey of tools. *Trans. on Embedded Computing Systems*, 7(3):1–53, 2008.

15 V. Yodaiken. Against Priority Inheritance. Technical report, Finite State Machine Labs, 2004.