Cache-related preemption delay via useful cache blocks: survey and redefinition

Sebastian Altmeyer, Claire Maiza Burguière

Angaben zur Veröffentlichung / Publication details:

Altmeyer, Sebastian, and Claire Maiza Burguière. 2011. "Cache-related preemption delay via useful cache blocks: survey and redefinition." *Journal of Systems Architecture* 57 (7): 707–19. https://doi.org/10.1016/j.sysarc.2010.08.006.

Cache-related preemption delay via useful cache blocks: Survey and redefinition ☆

Sebastian Altmeyer *, Claire Maiza Burguière

Compiler Design Lab, Saarland University, 66041 Saarbrücken, Germany

A B S T R A C T

Tasks in an embedded system are scheduled either preemptively or non-preemptively. In case of preemptive scheduling, interferences on the cache of the preempted and preempting task may extend the execution times. The corresponding delay is referred to as cache-related preemption delay (CRPD). Lee et al. [6] presented a CRPD analysis using the concept of useful cache block (UCB): a cache block is useful if it may be in the cache before a program point and may be reused after this point. If a preemption occurs at that point, the number of additional cache misses is bounded by the number of UCBs. An upper bound on the CRPD of the whole task is thus given by the program point with the largest set of UCBs. In this article, we provide a survey of the state of the art techniques to bound the CRPD, based on, but not limited to UCBs. Based on this survey we present an alternative definition of UCBs to improve the CRPD bounds substantially.

Keywords:
Context-switch costs
Preemptive scheduling
Cache memory
Static timing analysis
Hard real-time systems

1. Introduction

Tasks in an embedded system are scheduled preemptively or non-preemptively. A non-preemptive schedule offers a better predictability: the execution of each task can be analyzed separately, no impact of the execution of one task on the other tasks needs to be taken into account. Worst-case execution time (WCET) analyses are typically performed in such a setting. However, some task sets are only schedulable preemptively.

The major disadvantage of preemptive scheduling is the additional execution time due to preemption, referred to as context-switch costs (CSC). These context-switch costs are mainly the result of interferences on the cache of the preempted and preempting task. This fraction of the CSC is referred to as cache-related preemption delay (CRPD): cached memory blocks of a preempted task may be evicted due to the execution of the preempting task. If such a memory block is reused by the preempted task after preemption, an additional cache miss occurs. Thus, any attempt at determining the CRPD must consider the reuse of the available cache contents of the preempted task (UCB) or the damage done to this cache contents by the preempting task (ECB).

This article is composed of two parts. The first part provides a survey of the state of the art in the computation of the cache-related preemption delay—mainly relying on the concept of useful cache blocks. Based on this survey, we present an alternative definition of useful cache blocks to improve the CRPD bounds substantially. This alternative approach directly reuses results of the WCET analysis instead of analyzing the CRPD in complete isolation. By this, the number of additional cache misses contributing to the CRPD bound can be largely decreased.

In the remainder of the introduction, we provide the basic notion of caches and cache analyses, we give an example to explain the effect of a preemption on the cache, and we discuss the use of CRPD and WCET in the schedulability analysis.

1.1. Cache memory

Caches are fast but small memories storing frequently used memory blocks to close the increasing performance gap between processor and main memory. They are used as data, instruction or combined caches. An access to a memory block which is already in the cache is called a *cache hit*. An access to a memory block that is not cached, called a *cache miss*, causes the cache to load or store the data from/to the main memory.

Caches are divided into *cache lines*. A *cache line* is the basic unit to store parts of the memory. The memory again is divided into *memory blocks* of *line size* l contiguous bytes. The set of all memory blocks is denoted by M. The cache size s is thus given by the number of cache lines cl times the line size l. A set of n cache lines forms one *cache set*, where n is the *associativity* of the cache and determines the number of cache lines a specific memory block may reside in. The number of sets c is given by cl/n and the cache-set the memory block with address b maps to is given by $b \bmod c$.

Special cases are direct-mapped caches and fully-associative caches. In the first case, each cache line forms exactly one cache set

☆ This work was supported by ICT Project PREDATOR in the European Community's Seventh Framework Programme under grant Agreement No. 216008, by Transregional Collaborative Research Center AVACS of the German Research Council (DFG) and by ARTIST DESIGN NoE.

* Corresponding author.

E-mail addresses: altmeyer@cs.uni-saarland.de (S. Altmeyer), maiza@cs.uni-saarland.de (C. Maiza Burguière).

and there is exactly one position for each memory block ($n = 1$). In the second case, all cache lines together form one cache set and all memory blocks compete for all positions ($n = cl$). If the associativity is higher than 1 a *replacement policy* has to decide in which cache line of the cache set a memory block is stored, and, in case all cache lines are occupied, which memory block to remove. The most common replacement policies are:

First-in first-out (FIFO): The memory blocks are replaced in the order in which they were cached. This means, in case of a cache miss, the oldest element is removed; cache hits do not change the order.

Least recently used (LRU): The memory blocks are replaced in the reversed order in which they were used. This means, in case of a cache miss, the least-recently-used element is evicted, in case of a cache miss, the cached elements are—at least conceptually—reshuffled to maintain the order.

Pseudo least recently used (PLRU): PLRU approximates LRU at a lower complexity. It is implemented using a set of pointers always pointing to the element evicted next.

For an overview of the different replacement policies see Reineke [14].

The main concept behind caches is the exploitation of temporal and spatial locality. A memory block recently accessed is likely to be accessed again in the near future (temporal locality). Adjacent memory blocks are assumed to be accessed together (spatial locality). Although data and instruction caches profit from both phenomena, memory access patterns are vastly different. Instruction occurrences are stored at specific memory blocks and hence, cache accesses are bound to program points. This means, a cache block may only be reused, if the corresponding program point is executed again. Hence, the access patterns of the instruction caches have a very regular structure (following the structure of the control flow graph). In case of data caches, the accesses do not necessarily obey such a regular pattern. So, one memory block may be accessed several times within a sequential code segment with more than one access in between—which is not possible for the instruction cache.

1.2. Cache analysis

As a part of a timing analysis, a cache analysis aims to statically predict the cache behavior during the execution of a task. For this reason, the analysis classifies memory accesses as cache hits or cache misses. Due to input-dependent cache behavior and an unknown initial cache state, the outcome of this classification is not complete, i.e., not all memory accesses can be classified. To circumvent this problem, the concept of *may* and *must* information has been introduced to bound the cache contents from above and below. The may cache contains all memory blocks that may be cached at a given program point, i.e., where the analysis is unable to prove that the memory block is not cached. Vice versa, the must cache at a program point contains all memory blocks the analysis can prove to be cached. The use of this classification depends on the target architecture. In case of timing anomalies [9], a cache hit may lead to a longer execution than a cache miss. Thus, if the cache analysis is unable to safely classify a memory access, timing analysis has to take both possibilities into account: a cache hit and a cache miss. In case of architectures without timing anomalies, e.g., Arm7 processor, timing analysis considers all accesses to memory blocks not contained in the must cache as cache misses, since cache misses always lead to a longer execution time than cache hits.

Several different cache analyses have been proposed: Refs. [4,5,8,10,20,16], either for direct-mapped or n-way associative LRU caches, and for instruction or data caches.

In its simplest form, a cache analysis for direct-mapped caches is implemented as a program analysis keeping an abstract cache state at each program point. Such an abstract cache state contains for each cache set s either a set of memory blocks or the empty set $\emptyset$ indicating that the content of s could not be predicted. Encountering an access to a memory block m, the analysis stores m at the corresponding position in the abstract state (while replacing the prior content; see Fig. 1(a)). Two abstract cache states are combined by the use of intersection (must cache, Fig. 1(b)) or union sets (may cache, Fig. 1(c)).

For data caches, a value analysis [3] statically derives effective addresses of memory accesses (for instruction caches, the memory accesses are given by the addresses of the instructions, and thus, are fixed). If the value analysis, however, fails for some accesses, the precision of the cache analysis suffers. Even if the analysis is able to give a range for an access, for instance an access to an array component, the must-cache analysis can often not predict which block is cached. We denote a memory access for which no effective address can be statically determined a *dynamic memory access*.

1.3. Impact of a preemption on cache content—example

In this part, we demonstrate with an example how a preemption may affect the cache content and, thus, may cause additional cache misses. Suppose a sequence of memory accesses as presented in Fig. 2. There are five accesses (a–e) to five memory blocks of an instruction and/or data cache. In this example, the cache size is four lines and each line contains one memory block (direct-mapped cache). Memory blocks a to d are already cached before the execution. In case no preemption occurs (left part of Fig. 2), only one miss is observed to load the last cache block (e). In case a preemption occurs after the access to block b (right part of Fig. 2), the preempting task evicts three cache lines (blocks a, b

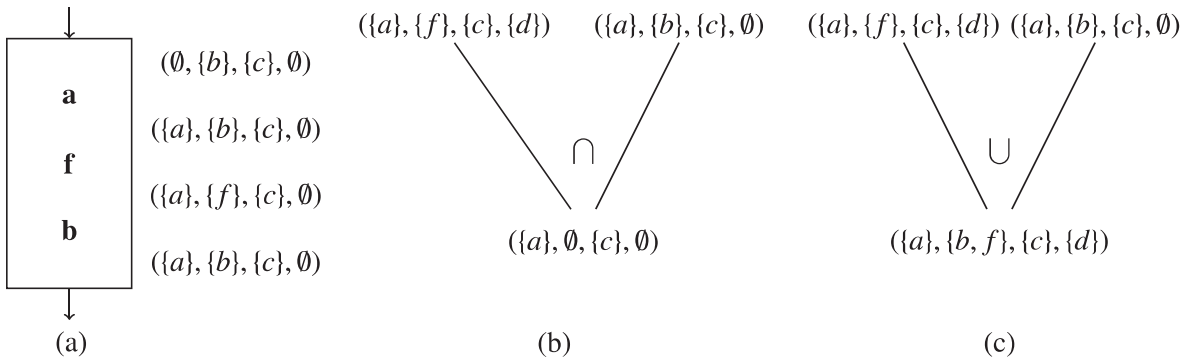


Fig. 1. Cache analysis: straight-line code (a) and control flow merge for must cache (b), and for may cache (c). In the figure, letters (a)–(f) denote addresses of different instructions.

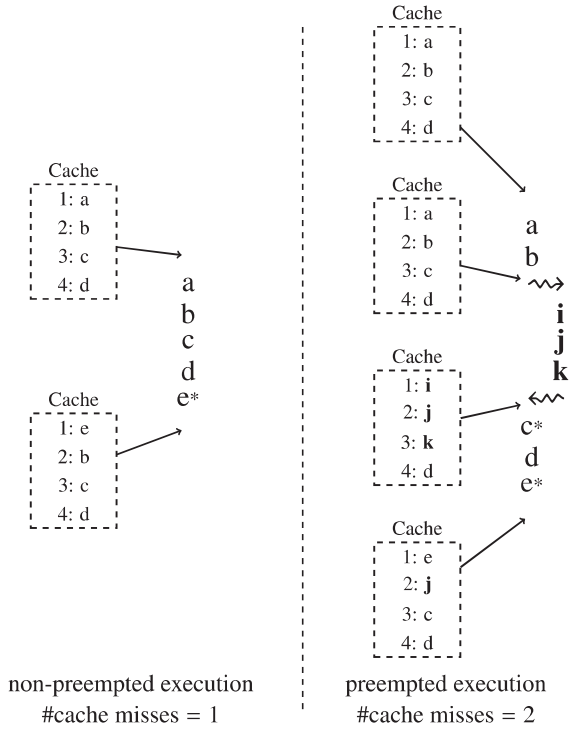


Fig. 2. A sequence of memory accesses (a–e) with and without preemption and the corresponding cache states. Each miss is marked by $*$.

and c). After the preemption, the preempted task needs to reload cache block c which is evicted and would be reused otherwise. Cache block d has not been evicted by the execution of the preempting task. Loading the last cache block of the sequence (e) is not a cache miss due to the preemption as it was not in the cache before. So, in case of preemption, one additional miss occurs. The preemption cost has to account for this miss penalty.

1.4. CRPD and WCET

Within schedulability analysis, the CRPD is always added to the WCET. Hence, it is sufficient to guarantee that $n \times \text{CRPD} + \text{WCET}$, where n is the maximal number of preemptions, bounds the actual execution time under preemption—instead of providing a bound on the CRPD in isolation. The idea of a weaker guarantee was already present in the initial work by Lee et al. [6]—but only for dynamic memory accesses to the data cache. So, it was not thoroughly applied to the concept of UCB. Note that this weaker guarantee is the basis for the concept of DC-UCB as presented in the second part of this paper.

2. State of the art

2.1. Bounding the CRPD

In this section we focus on the derivation of upper bounds on the CRPD. We explain the concept of useful cache blocks (UCB) and of evicting cache blocks (ECB). Furthermore, we show how they can be used separately or combined to derive this bound. We first focus on the CRPD computation for direct-mapped caches. Then, for set-associative caches, we show that this computation needs to be carefully applied to the LRU policy and can not be adapted to the FIFO and PLRU policies. Note that all definitions and all formulas to bound the CRPD given in this section are valid for instruction and data caches. However, a CRPD analysis is only feasible for processors without timing anomalies. In such a case, the number of additional misses times the cache reload time provides a safe bound on the additional execution time.

2.1.1. Bounding the CRPD for direct-mapped caches

The cache-related preemption delay denotes the additional execution time due to cache misses caused by preemption. As shown in the example in Fig. 2, such cache misses occur, when the preempting task evicts cache blocks of the preempted task that otherwise would be reused later. Upper bounds on the CRPD can be derived from two directions:

1. bounding the worst-case effect on the preempted task: the worst-case number of cache blocks for which an access after a preemption point may cause an additional miss, i.e., blocks that are cached before preemption and reused afterwards;
2. or bounding the effect of the preempting task: the number of distinct memory blocks accessed during the execution of the preempting task—potentially evicting cache blocks of the preempted task.

Depending on the structure of the tasks, results of both analyses may widely differ.

2.1.1.1. Analysis of the preempted task. For the analysis of the effect on the preempted task, Lee et al. [6] introduced the concept of useful cache block:

Definition 1 (*Useful cache block (UCB)*). A memory block m is called a useful cache block at program point P, if

(a) m may be cached at P and
(b) m may be reused at program point Q that may be reached from P without eviction of m on this path.

In the case of preemption at program point P, only the memory blocks that (a) are cached and (b) will be reused, may cause additional reloads. Hence, the number of UCBs at program point P gives an upper bound on the number of additional reloads due to a preemption at P. A global bound on the CRPD of the whole task is determined by the program point with the highest number of UCBs. The following formula gives an upper bound on the CRPD for direct-mapped caches when UCBs are used to derive this upper bound (where c denotes the number of cache sets):

$$\text{CRPD}_{\text{UCB}} = \text{CRT} \cdot |\{s_i | \exists m \in \text{UCB} : m \bmod c = s_i\}| \quad (1)$$

The CRPD_{UCB} is bounded by the cache reload time (CRT), i.e., the time needed to load a cache block, times the number of sets, which at least one UCB maps to (see [6]). Note that the CRPD bounds denote the additional delay for one preemption. In case of several preemptions, the CRPD bound must be added to the WCET as often as preemption might occur.

2.1.1.2. Analysis of the preempting task. The worst-case impact of the preempting task is given by the number of cache blocks this task may evict during preemption. Obviously, each memory block possibly-cached during the execution of the preempting task may evict a cache block of the preempted one:

Definition 2 (*Evicting cache blocks (ECB)*). A memory block of the preempting task is called an evicting cache block, if it may be accessed during the execution of the preempting task.

Accessing an evicting cache block (ECB) in the preempting task may evict a cache block of the preempted task. Tomiyama and Dutt [19] proposed to use only the number of ECBs to bound the CRPD.

The following formula gives an upper bound on the CRPD for direct-mapped caches when ECBs are used to derive this upper bound:

$$\mathrm{CRPD}_{\mathrm{ECB}} = \mathrm{CRT} \cdot |\{s_i | \exists m \in \mathrm{ECB} : m \bmod c = s_i\}| \tag{2}$$

The CRPD is bounded by the cache reload time times the number of sets, which at least one ECB maps to (see [19]). Note that in case of nested preemption the set ECB in the formula is the union of all ECB sets of the preempting tasks, see [19,15].

2.1.1.3. Analysis of preempting task and preempted task. Negi et al. [11] and Tan and Mooney [17] proposed to combine the number of ECBs and UCBs to improve the CRPD bounds; only useful cache blocks that are actually evicted by an evicting cache block may contribute to the CRPD. The following formula gives a precise upper bound on the CRPD for direct-mapped caches using UCBs and ECBs:

$$\mathrm{CRPD}_{\mathrm{UCB\&ECB}} = \mathrm{CRT} \cdot |\{s_i | \exists m \in \mathrm{UCB} : m \bmod c = s_i \wedge \exists m' \in \mathrm{ECB} : m' \bmod c = s_i\}| \tag{3}$$

The $\mathrm{CRPD}_{\mathrm{UCBECB}}$ is bounded by the cache reload time times the number of sets, which at least one UCB and one ECB map to (see [11,17]).

The derivation of the sets of UCBs for direct-mapped caches, corresponding to the definition presented in this section, is detailed in Section 2.2.1. The derivation of the sets of ECBs is presented in Section 2.2.4. As explained in Section 2.1.2, for set-associative caches, the derivation of an upper bound on the CRPD, based on UCBs and/or ECBs, is not always possible.

2.1.2. Bounding the CRPD for set-associative instruction caches

The definitions of UCBs and ECBs are not restricted to some types of cache architectures, but are expected to apply for set-associative caches with any replacement policy. Of course, whether a block is useful or not depends on the particular cache architecture, i.e., its associativity and replacement policy. In addition, for set-associative caches, the CRPD computation based on UCBs and on ECBs differs from one replacement policy to another.

2.1.2.1. CRPD for LRU set-associative caches. An upper bound on the CRPD for LRU set-associative caches has been computed using only the number of UCBs [7] or using a combination of the number of UCBs and the number of ECBs [17]. In the first case, Lee et al. [7] compute the bound on the CRPD in the same manner as for direct-mapped caches, by only taking into account the number of useful cache blocks. The difference comes from the fact that they compute it for each cache set independently and the overall sum gives the global bound on the number of misses due to preemption. The following formula gives an upper bound on the CRPD for LRU caches by using solely the UCBs:

$$\mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{UCB}} = \mathrm{CRT} \cdot \sum_{s=1}^{c} |\mathrm{UCB}(s)| \tag{4}$$

where $\mathrm{UCB}(s)$ denotes the set of UCBs mapping to cache set s.

Tan and Mooney [17] applied the idea to use the preempting task(s) in the analysis of set-associative caches. They compute for each cache set the number of evicting cache blocks ($\mathrm{ECB}(s)$). According to their approach, the cache-related preemption delay for each program point is given by the formula:

$$\mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{MIN}} = \sum_{s=1}^{c} \mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{MIN}}(s) \tag{5}$$

where

$$\mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{MIN}}(s) = \mathrm{CRT} \cdot \min(|\mathrm{UCB}(s)|, |\mathrm{ECB}(s)|, n) \tag{6}$$

where $\mathrm{UCB}(s)$ and $\mathrm{ECB}(s)$ denote the sets of UCBs and ECBs, respectively, mapping to cache set s and n is the number of ways (associativity). Note that depending on the UCB analysis, the number of UCBs may exceed the associativity.

The computation of the overall CRPD bound of the whole task is given by the maximum local CRPD—as for direct-mapped caches. Again, a CRPD bound is only valid for a single preemption. In case of multiple preemption the CRPD bound has to be added to the WCET as often as preemption occurs.

However, we noticed that this minimum function may underestimate the number of additional misses.

Observation 1. In the case of LRU-replacement policy, the number of ECBs can not be used as a safe upper bound on the number of additional misses due to preemptions.

Let us use the CFG of Fig. 3. The basic block contains instructions stored in four memory blocks (a–d). These memory blocks are mapped to the same cache set.

Therefore, the set contains all these blocks at the end of the execution of this basic block: this 4-way cache set is completely filled. As this basic block forms the body of a loop, these memory blocks are useful. Suppose a preemption occurs between the end point of the basic block and its next execution. Suppose in addition that the preempting task uses a memory block that maps to the same set (memory block **x**). In that case, this block evicts one useful cache block: Using the minimum function, only one additional miss is taken into account for this memory set ($\min(4, 1, 4) = 1$). Block a is evicted because it is the least recently used. After preemption, this block is reloaded: it evicts block b. The reload of block b will evict the next one (c). Finally all blocks are evicted and reloaded. The preempting task uses only one block mapped to this set. However, it causes four additional misses (as many as the number of ways). In this example, the number of UCBs and the number of ways (associativity) are upper bounds on the number of misses, but the minimum function gives an underestimation of the number of additional misses.

Instead of using the formula by Tan and Mooney [17], the results from the CRPD computation via UCB and via ECB can be combined in a straight-forward manner:

$$\mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{UCBECB}} = \sum_{s=1}^{c} \mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{UCBECB}}(s) \tag{7}$$

where

$$\mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{UCBECB}}(s) = \begin{cases} 0 & \text{if } \mathrm{ECB}(s) = \emptyset \\ \mathrm{CRT} \cdot \min(|\mathrm{UCB}(s)|, n) & \text{otherwise} \end{cases} \tag{8}$$

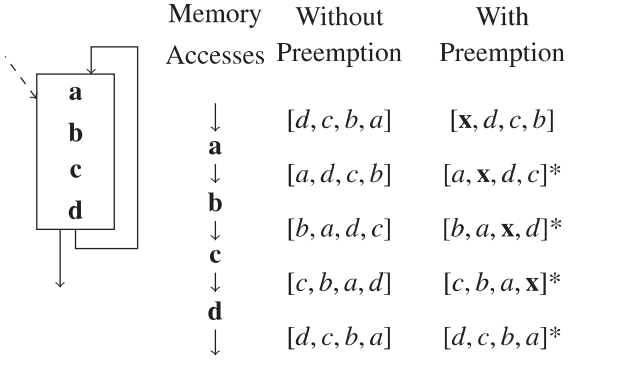


Fig. 3. Evolution of the cache content in case of LRU replacement. The first column shows the sequence of memory accesses. The second column is the cache-set content (4-way). The last column is the cache-set content after preemption. Each miss is marked by $*$. Blocks a, b, c and d are useful before this memory access sequence.

Again, UCB(s) and ECB(s) denote the sets of UCBs and ECBs, respectively, mapping to cache set s. This function refines the CRPD in case no ECB maps to a set s and in case the number of UCBs in a set s is higher than the associativity. Note that the CRPD is always bounded by the associativity and the number of UCBs. Staschulat and Ernst [16] derive the sets of UCBs and ECBs for LRU set-associative caches. They use the same equation as for the direct-mapped cache to derive the upper bound on the CRPD. However, the intersection of UCB and ECB (as used for direct-mapped caches) may underestimate the CRPD in case of set-associative caches, as shown above.

Note that recently a further improvement on the CRPD bound for LRU based on UCB and ECB has been presented by Altmeyer et al. [1].

2.1.2.2. CRPD for FIFO or PLRU. Often, related articles implicitly assume LRU-replacement policy in case of set-associative caches. The extension to FIFO and PLRU is then claimed to be straight-forward. However, as the following observation states, UCB and ECB approaches are unsuitable for these replacement policies in general.

Observation 2. In case of FIFO or PLRU replacement, neither the number of UCBs nor the number of ECBs nor the associativity are safe upper bounds on the number of additional misses due to preemption.

Fig. 4 shows that the number of additional misses may be higher than the number of UCBs and even the associativity. It depicts a sequence of accesses to a 2-way set-associative FIFO cache. The preemption occurs before the presented sequence: blocks a and b are useful, both are evicted and the final content of this set after preemption is $[y, x]$. The number of misses in the first case is 2 and 5 in case of preemption. The number of additional misses (3) is greater than the number of UCBs (2), the number of ECBs (2) and the associativity (2). So, these numbers can not be used as an upper bound on the number of additional misses when the FIFO replacement strategy is used.

A similar counter example for PLRU replacement strategy can be found in [2]. In contrast to LRU, for FIFO and PLRU, the difference in the number of misses along the same sequence of memory accesses but starting with two different cache states cannot be bounded by a constant (see the concept of sensitivity in [14]). In case of a preemption the cache state is modified at one execution point of the preempted task by the execution of the preempting task. Thus, for both strategies, neither the number of UCBs, nor the number of ECBs can be used to compute an upper bound on the CRPD: UCB and ECB derivation are not suitable for these replacement strategies.

Memory Accesses	Without Preemption	With Preemption
↓	$[b, a]$	$[\mathbf{y}, \mathbf{x}]$
a		
↓	$[b, a]$	$[a, \mathbf{y}]$*
e		
↓	$[e, b]$*	$[e, a]$*
b		
↓	$[e, b]$	$[b, e]$*
c		
↓	$[c, e]$*	$[c, b]$*
e		
↓	$[c, e]$	$[e, c]$*
	2 misses	5 misses

Fig. 4. Evolution of the cache content in case of FIFO replacement. The first column shows the sequence of memory accesses. The second column is the cache-set content (2-way). The last column is the cache-set content after preemption. Each miss is marked by $*$. Blocks a and b are useful before this memory access sequence.

2.2. Data-flow analysis

In this section, we describe how to derive the set of useful cache blocks needed for the CRPD computation. We start with the original analysis for direct-mapped instruction caches as introduced by Lee et al. [6], followed by improvements by Negi et al. [11] and Staschulat and Ernst [15]. The extensions to set-associative and data caches are presented at the end of this section.

The UCB analysis and all of its extensions are implemented as data-flow analyses. Programs under examination are represented as control flow graphs (CFG). Nodes of the CFGs are basic blocks: maximal sequences of instructions with exactly one entry and one exit point. Therefore, if one instruction of a basic block is executed, so are all others. The edges of the CFG represent the possible control flows.

Definition 3 (*Control flow graph*). An analyzed program $\mathbb{P}$ is represented as a control flow graph $CFG = (V, E, s, e)$ where $V = \{B_1, \ldots, B_n\}$ denotes the set of basic blocks B_i of $\mathbb{P}$ and $E \in V \times V$ the corresponding edges connecting them. The start node is denoted by s and the end node by e, respectively.

2.2.1. UCB for direct-mapped instruction caches

Since a task may be preempted at any position, the set of useful cache blocks has to be computed at each instruction in the control flow graph. However, in the case of instruction caches, it is sufficient to derive the set of UCBs only at the basic block level (see [6]).

Theorem 1. *The set of UCBs does not change within a basic block and, by this, the number of UCBs is constant within a basic block.*

The theorem follows directly from the possible access pattern for instruction caches and the coupling of instruction occurrences (and hence, memory accesses) to program points (see Section 1.2). To have different sets of UCBs at different instructions of a basic block B, a memory block must be evicted and reused within B—which is not possible in case of instruction caches. Hence, the set of UCBs is the same for all instructions of a basic block. Note that we simplify the cache in a way such that each memory block contains exactly one instruction. This simplification can be found implicitly in the related work, too. In case a memory block contains several instructions, the number of UCBs must be increased by one to account for the additional cache miss to reload the currently used memory block. The simplification, however, does not influence the correctness of the theorem.

As a consequence of the theorem, the set of UCBs only needs to be computed on the basic block level in case of instruction caches.

2.2.1.1. Reaching memory blocks/live memory blocks. The computation of the set of UCBs uses two data-flow analyses: The first one to compute all memory blocks where the condition "m may be cached at P" (condition a) of Definition 1) holds and the second one for condition "m may be reused at program point Q that may be reached from P without eviction of m on this path" (condition b) of Definition 1). The intersection of both sets finally delivers the set of UCBs. Note that the first analysis resembles a may cache analysis as described in Section 1.2. However, since CRPD and WCET are computed separately, two different analyses may be used.

The *reaching memory blocks (RMB)* at program point P are all cached memory blocks at P. The *live memory blocks (LMB)* at program

point P are all memory blocks that may be used on some path starting at P without being evicted on this path. The analysis of both sets are strongly related to the well-known program analyses for live variables and reaching definitions (see [12]). Both can be implemented as fixed-point iterations on the control flow graph.

In the following, we will describe the RMB and LMB analyses based on Lee et al. [6]. However, we do not stick to their notation, but use a simplified one in order to ease the comparison of the different methods. The set of memory blocks is denoted by M and the cache contents of a cache set is a subset of M. Hence, the set of all cache-set states is the powerset of M, denoted by 2^M. The contents of the whole cache, and thus, the domain for the analyses is then given by the cartesian product of c sets of cache-set states:

$$\mathbb{D}_{\text{Lee}} := \overbrace{2^M \times 2^M \times 2^M \times \cdots \times 2^M}^{c}$$

Note that for direct-mapped caches, each cache set contains exactly one memory block. However, the analysis can only compute an approximation of the actual cache content and thus, maintains a set of memory blocks for each cache set.

The combine-operator of this domain, invoked to merge flow information coming from different control flow paths, is defined as the pointwise union on the components of its operands:

$$(C_1, C_2, \ldots, C_c) \sqcup (C'_1, C'_2, \ldots, C'_c) = (C_1 \cup C'_1, C_2 \cup C'_2, \ldots, C_c \cup C'_c)$$

where $C_i \in 2^M$ is the set of possibly-cached elements at cache set i.

The analysis of reaching memory blocks is defined as a forward analysis. Hence, the abstract cache state at the entry of basic block B is determined by the combination of all abstract cache states of its predecessors:

$$RMB_{in}(B) = \bigsqcup_{\text{predecessor } B'} RMB_{out}(B')$$

The update of the abstract cache state at B resembles the cache behavior during the execution of B. This means that the memory block at cache set i is replaced by the last memory block m that is accessed within B and maps to cache set i. If there is no such memory block, the abstract cache state at cache set i remains unchanged:

$$RMB_{out}(B) = last\_acc(RMB_{in}(B))$$

where

$$last\_acc((C_1, C_2, \ldots, C_c)) = ((last\_acc_1(C_1), last\_acc_2(C_2), \ldots, last\_acc_c(C_c))$$

and

$$last\_acc_i(C_i) = \begin{cases} \{m\} & \text{if } m \text{ is the } \textit{last} \text{ memory block in} \\ & B \text{ that maps to cache set } i \\ C_i & \text{if there is no such memory block} \end{cases}$$

The analysis of live memory blocks is a backward analysis—it can be seen as the reverse of the former one. Hence, the abstract cache state after basic block B is determined by the combination of all abstract cache states of its successors (instead of predecessors):

$$LMB_{out}(B) = \bigsqcup_{\text{successor } B'} LMB_{in}(B')$$

The update of the abstract cache state at B resembles the cache behavior during the execution of B in reverse order. This means that the memory block at cache set i is replaced by the first—instead of the last—memory block m within B that maps to cache set i. If there is no such memory block, the abstract cache state at cache set i remains unchanged:

$$LMB_{in}(B) = first\_acc(LMB_{out}(B))$$

where

$$first\_acc((C_1, C_2, \ldots, C_c)) = ((first\_acc_1(C_1), first\_acc_2(C_2), \ldots, first\_acc_c(C_c))$$

and

$$first\_acc_i(C_i) = \begin{cases} \{m\} & \text{if } m \text{ is the } \textit{first} \text{ memory block in} \\ & B \text{ that maps to cache set } i \\ C_i & \text{if there is no such memory block} \end{cases}$$

In both cases (RMB and LMB), the fixed-point iteration based on the above equation starts with initially empty cache states, i.e., a c-tuple of empty sets $(\emptyset, \emptyset, \ldots, \emptyset)$. Fig. 5 shows a control flow graph, and the resulting sets of RMBs, LMBs and UCBs are shown in Table 1. The number in the upper-right corner of the boxes denote the indices i of the corresponding basic blocks. Remember that a memory block is useful when it is contained in both the RMB_{in} and the LMB_{in} sets.

2.2.1.2. Set of cache states. Lee's (non-relational) domain is given as a tuple of sets; for each cache set i, a set of UCBs possibly residing at i is given. Since not all combinations of elements in these sets

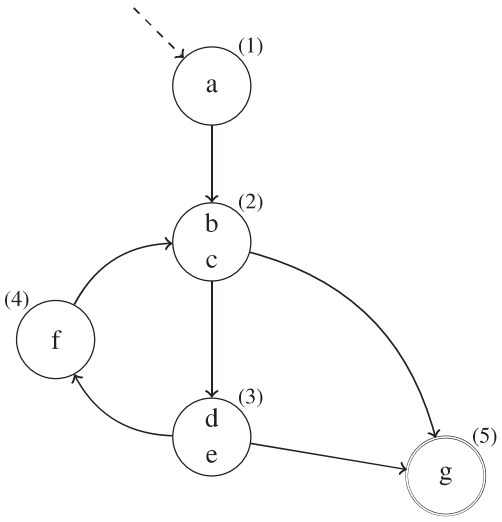


Fig. 5. Example control flow graph.

Table 1
Resulting RMB, LMB and UCB sets for the CFG shown in Fig. 5; a and e map to cache set 1, b and f to set 2, c and g to set 3 and d to set 4.

i	$RMB(B_i)$	$LMB(B_i)$
in		
1	$(\emptyset, \emptyset, \emptyset, \emptyset)$	$(\{a\}, \{b\}, \{c\}, \{d\})$
2	$(\{a, e\}, \{f\}, \{c\}, \{d\})$	$(\{e\}, \{b\}, \{c\}, \{d\})$
3	$(\{a, e\}, \{b\}, \{c\}, \{d\})$	$(\{e\}, \{f\}, \{c, g\}, \{d\})$
4	$(\{e\}, \{b\}, \{c\}, \{d\})$	$(\{e\}, \{f\}, \{c\}, \{d\})$
5	$(\{a, e\}, \{b\}, \{c\}, \{d\})$	$(\emptyset, \emptyset, \{g\}, \emptyset)$
out		
1	$(\{a\}, \emptyset, \emptyset, \emptyset)$	$(\{e\}, \{b\}, \{c\}, \{d\})$
2	$(\{a, e\}, \{b\}, \{c\}, \{d\})$	$(\{e\}, \{f\}, \{c, g\}, \{d\})$
3	$(\{e\}, \{b\}, \{c\}, \{d\})$	$(\{e\}, \{f\}, \{c, g\}, \{d\})$
4	$(\{e\}, \{f\}, \{c\}, \{d\})$	$(\{e\}, \{b\}, \{c\}, \{d\})$
5	$(\{a, e\}, \{b\}, \{g\}, \{d\})$	$(\emptyset, \emptyset, \emptyset, \emptyset)$

i	$UCBs(B_i)$
1	$(\emptyset, \emptyset, \emptyset, \emptyset)$
2	$(\{e\}, \emptyset, \{c\}, \{d\})$
3	$(\{e\}, \emptyset, \{c\}, \{d\})$
4	$(\{e\}, \emptyset, \{c\}, \{d\})$
5	$(\emptyset, \emptyset, \emptyset, \emptyset)$

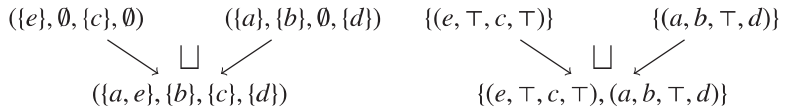


Fig. 6. Overapproximation due to Lee's domain (left part) and improved domain by Negi (right part).

actually occur, the domain over-approximates the actual number of UCBs (see left part of Fig. 6). Negi et al. [11] therefore introduce a new analysis based on a relational domain to derive more precise information about the set of UCBs. They represent the abstract cache states in the following manner. The abstract cache content of cache set i is given by $C_i^N \in M_\top := M \cup \{\top\}$. If $C_i^N = \top$, cache set i is either empty or the analysis was unable to predict the content of cache set i. The domain of the entire cache state is given by the cartesian product of the domains of the cache-set states, i.e.,

$$CS := \overbrace{M_\top \times M_\top \times M_\top \times \cdots \times M_\top}^{c}$$

However, several distinct cache states may occur at one program point during program execution. So, the domain of the analysis is given as the powerset of the domain of the cache states CS: $\mathbb{D}_{\text{Negi}} := 2^{CS}$. The combine-operator of the new domain is set-union.

The difference between Lee's and Negi's representations can be seen in Fig. 6. The number of UCBs of the incoming sets is 2 on the left and 3 on the right path. In Lee's approach, pairwise union for each cache set is applied. The maximal number of UCBs in the resulting set is 4, i.e., each cache set may contain one UCB. In Negi's approach, the incoming cache states are kept separate. So, the maximal number of UCBs is 3.

2.2.1.3. Trade-off precision/run-time. Negi's representation of the cache content enhances Lee's original approach. However, the complexity of computing the set of UCBs is much higher. Staschulat and Ernst [16] propose a tradeoff between complexity and tightness. The aim is to obtain a scalable precision by limiting the number of cache states thus reducing the complexity: They propose to merge cache states which are almost equal. This is done by modifying the RMB_{in} computation proposed by Negi et al. [11]:

$$RMB_{in}(B) = bound_z(\underbrace{\bigsqcup_{\text{predecessor } B'} RMB_{out}(B')}_{CStates})$$

where $bound_z(CStates)$ reduces the number of total cache states of $CStates$ to z. The reducing function $bound_z$ chooses two cache states of minimal distance and merges them: this is repeated until the number of states in $CStates$ equals z. The two elements with the minimum distance are replaced by the merged cache state. The distance metric is a function that delivers the difference of two cache states. For example, Staschulat and Ernst [16] use the number of cache sets with different cache contents as the distance between two cache states.

2.2.2. Set-associative instruction caches

Embedded systems often use set-associative caches. Lee et al. [6] extended their UCB analysis to handle them. Furthermore, they assume the LRU replacement policy—although this is not stated explicitly in the paper.

A cache set of an n-way set-associative LRU cache contains up to n cache blocks. One state of a cache set can be described by a n-tuple of memory blocks:

$$S := \overbrace{M \times M \times \cdots \times M}^{n}$$

All possible states of cache set i are given by a set of such n-tuples ($C_i^S \in 2^S$). The domain of the UCB analysis for set-associative caches is then given by the cartesian product of c sets of cache-set states:

$$\mathbb{D} := \overbrace{2^S \times 2^S \times 2^S \times \cdots \times 2^S}^{c}$$

Again, the UCB analysis uses the sets of reaching memory blocks (RMB) and live memory blocks (LMB) and the computation of these sets resembles the cache behavior. According to Lee's convention, the leftmost memory block in the tuple is the oldest, the rightmost memory block the youngest cached element. If one memory block m_B mapping to cache set i is accessed within basic block B, m_B is set to the rightmost position and all other elements are shifted one to the left. To explain this computation, we assume that the memory blocks m_1 to m_4 are cached in set i, i.e., $C_i^S = \{(m_4, m_3, m_2, m_1)\}$. The access of block m_B then changes the cache content to $\{(m_3, m_2, m_1, m_B)\}$.

In general, the number of memory blocks mapping to cache set i within basic block B determines how far former elements are shifted to the left and how many new memory blocks are inserted on the right of cache contents C_i^S. Note that for the computation of LMB, the cache behavior is simulated in reverse execution order.

As Staschulat and Ernst [16] noted, the computation of LMB and RMB described above contains a flaw: the same memory block may be contained up to n times in the abstract state. An element is inserted into C_i^S no matter if it is already contained or not. To correct the former computation, Staschulat and Ernst proposed to mimic the LRU-replacement policy in detail. If none of the accessed elements are contained in C_i^S, the RMB/LMB sets are computed as above. In case a memory block m_B accessed within B is already contained in C_i^S, this element is put into the rightmost position, and only elements from the rightmost up to m_B's former position are shifted to the left.

2.2.3. Data caches

UCB computation for data caches has been limited to statically addressed data (see [6,17]). Lee et al. assume cache analysis to be unable to handle dynamic memory accesses,[1] which causes these accesses to be treated as cache misses. Therefore they do not contribute additional costs to the CRPD—as explained in Section 1.4. As far as we know, the only work taking into account dynamic addressing in the context of CRPD analysis has been done by Ramaprasad and Mueller [13]. Their approach, however, does not use the concept of UCBs. Instead, they employ access patterns to compute the CRPD. Note that the set of UCBs for data caches (static addressing) has to be computed for each instruction. As already discussed in Section 1.1, data accesses do not occur isomorphic to the control flow graph. Hence, the set of UCBs may change at the instruction level and not only at the basic block level.

2.2.4. Evicting cache block analysis

For the sake of completeness, we sketch a prototypical ECB analysis. Note that, in contrast to the UCB analysis, the ECB analysis is applied to the preempting task and not to the preempted task. The set of evicting cache blocks (ECBs) is determined as the set of memory blocks possibly-cached during the execution of the preempting task [19]. It can be computed using the analysis of reaching memory blocks or the analysis of live memory blocks. Both analyses collect all cached memory blocks of the task. Thus, the fi-

[1] Remember that a dynamic memory access denotes a memory access for which no effective address can be derived statically.

nal set of ECBs is given by the set of all elements contained in the RMB at the end node e (or by the set of elements contained in the LMB at the first node s)—no matter which cache state representation was used to compute them.

2.3. Summary

Starting with Lee et al. [6], in over 10 years of research the computation of the bounds on the CRPD was successively improved. Two different approaches can be identified: analyzing the preempted task [6] and analyzing the preempting task [19]. Both approaches have been combined [11,17] and a more precise notion of abstract cache states [11,16] was introduced to improve the bounds on the CRPD.

For the analysis of set-associative caches, already Lee's original paper contained the basic ideas, which were corrected by Staschulat and Ernst [16]. The corresponding CRPD bound, however, is only valid for the LRU-replacement policy and not for FIFO or PLRU—as we have shown in Section 2.1.2.2.

Lee's UCB analysis for data caches only takes into account statically fixed data accesses—because their timing analysis treats all other accesses as cache misses. Thus, the derived bound does not bound the number of additional cache misses due to preemption, but only the number of additional cache misses due to preemption *not taken into account by the timing analysis*. Nevertheless, the computed bound is still sound. As we will see in the next part, the same idea can be applied to the whole UCB analysis in order to improve the CRPD bounds.

3. Redefinition of useful cache blocks

In this part, we enhance the computation of the bound on the CRPD by introducing a new notion of useful cache block. The notion of evicting cache block is not modified and can be used in combination with the new approach.

3.1. Definitely-cached useful cache block

An over-approximation of the cache contents (memory blocks that *may* be in cache) is used by former UCB analyses to derive a safe upper bound on the CRPD. This over-approximation of the cache contents is referred to as *may cache*. As presented in Section 1.2, the WCET analysis also uses an over-approximation of the cache contents to predict the number of cache misses. Remember that determining cache misses is sufficient for processors without timing anomalies.

Furthermore, WCET analysis uses an under-approximation of the cache content to predict the number of cache hits (memory blocks that *must* be in cache also referred to as *must cache*). Thus, only blocks contained in the may cache and the must cache are counted as cache hits. All blocks not contained in the must cache are counted as cache misses—even if they are in the may cache. For each access to these blocks, the cache reload time is added to the bound on the worst-case execution time (WCET).

Some of these blocks, contained in the may cache but not in the must cache, are useful and considered a miss by the UCB analysis. Thus, during schedulability analysis, these accesses are counted twice as a miss: as part of the CRPD bound and as part as the WCET bound. So, treating timing analysis and CRPD analysis separately, the over-approximation on both sides accumulates and introduces a high degree of pessimism.

In this section we introduce the notion of definitely-cached UCB excluding those misses that are counted twice. Then, we show the soundness of the DC-UCB approach: the bound on the CRPD computed using DC-UCB is safe when used in combination with a bound on the WCET.

3.1.1. Bounding the number of additional misses

For schedulability analysis, a bound on the CRPD is always used in combination with a bound on the WCET. Thus, we can remove from the set of UCBs all accesses already counted as a miss by the timing analysis. This is given by the notion of definitely-cached UCB.

Definition 4 (*Definitely-cached UCB (DC-UCB)*). A memory block m is called a definitely-cached UCB at program point P, if

(a) m *must* be cached at P, and,
(b) m may be reused at program point Q that may be reached from P and *must* be cached along the path to its reuse.

The DC-UCB analysis gives, for a given program point, an upper bound on the set of additional misses *not taken into account in the WCET analysis* when used in combination. The set of DC-UCBs might not give an upper bound on the CRPD. However, this bound is safe when combined with the bound on the WCET delivered by timing analysis: the over-approximation of the execution-time bound subsumes the possible under-approximation of the CRPD.

Note that Lee et al. [7] implicitly use the notion of DC-UCB in UCB analysis of data caches: they focus only on static addressing of data accesses because dynamic ones are considered as cache misses by their timing analysis. Hence, the computed bound only denotes the additional cache misses due to preemption.

3.1.1.1. Soundness. The combination of over-approximation in the timing analysis and in the conventional notion of useful cache blocks counts some potential cache reloads twice. The adapted definition excludes these reloads: So, they are only counted as part of the execution-time bound. The context-switch costs, thus, may be an under-approximation of the real costs but are a sound over-approximation when combined with the results of the timing analysis.

This can be seen by comparing (i) the overall number of cache misses that might occur during task execution with (ii) the number of cache misses the analyses (timing analysis and DC-UCB analysis) take into account. If our approach accounts for at least the number of actual cache misses, i.e., the number of cache misses that occur during the execution of the task, it can be considered to derive a safe over-approximation. Remember that using (DC-)UCBs to compute context-switch costs is restricted to processors not exhibiting timing anomalies. Hence, it is sufficient to prove that the number of misses is safely bounded in order to prove the correctness of the DC-UCB analysis.

Consider the set $\mathbb{A}$ of memory accesses that occur during the non-preemptive execution of a task. These accesses are either cache hits $\mathbb{H}$ or cache misses $\mathbb{M}$. By construction, the must-cache analysis classifies a subset of these accesses as cache hits, $\mathbb{H}_{must} \subseteq \mathbb{H}$. All other accesses are taken into account as cache misses, $\mathbb{M}_{must} = \mathbb{A} \setminus \mathbb{H}_{must}$. Hence: $\mathbb{H} \cup \mathbb{M} = \mathbb{A} = \mathbb{H}_{must} \cup \mathbb{M}_{must}$. The set of additional cache misses due to preemption is denoted by $\mathbb{P} \subseteq \mathbb{H}$. By definition, the set $\mathbb{P}'$ obtained by our analysis is an over-approximation of $\mathbb{P}$ restricted to elements of the set $\mathbb{H}_{must}$: $\mathbb{P}' \supseteq \mathbb{P} \cap \mathbb{H}_{must}$. By removing this set $\mathbb{P}'$ from the set of cache hits classified by the must cache, we get: $\mathbb{H}_{must} \setminus \mathbb{P}' \subseteq \mathbb{H}_{must} \setminus (\mathbb{P} \cap \mathbb{H}_{must}) = \mathbb{H}_{must} \setminus \mathbb{P}$. Since $\mathbb{H}_{must} \subseteq \mathbb{H}$, the set of cache hits classified by our method is a subset of the actual set of cache hits, we get: $\mathbb{H}_{must} \setminus \mathbb{P}' \subseteq \mathbb{H} \setminus \mathbb{P}$. This shows that our approach under-approximates the set of cache hits under preemption: The number (i) of actual cache misses, given by $(\mathbb{M} \cup \mathbb{P})$, is a subset of the number

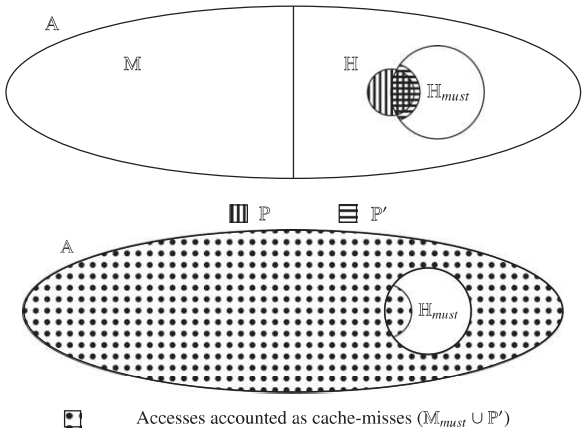


Fig. 7. Set of cache hits, $\mathbb{H}$, set of cache misses, $\mathbb{M}$, set of cache hits approximated by must cache, $\mathbb{H}_{must}$, set of cache misses due to preemption, $\mathbb{P}$, and set of cache misses, $\mathbb{P}'$, derived by our analysis. The dotted part in the second graph denotes all accesses taken into account as cache misses by our approach.

(ii) of cache misses taken into account by our analysis, given by $(\mathbb{M}_{must} \cup \mathbb{P}')$.

By this, we showed that our approach safely bounds the cache-related preemption delay when combined with an upper bound on the WCET. Fig. 7 illustrates the different sets and their relation.

3.1.2. Bounding the CRPD

Remember that we bound the CRPD taking into account the WCET analysis. Remember also that the CRPD bounds denote the additional delay for one preemption. In case of several preemptions, the CRPD bound must be added to the WCET as often as preemption might occur. Since neither the ECB definition, nor the computation of the bound on the CRPD based only on ECBs are modified by the new approach, we can simply replace UCB by DC-UCB in all former CRPD formulas given in Section 2.1.2.

Note that for FIFO and PLRU caches, DC-UCB does not help to bound the CRPD: as we showed it in Section 2.1.2, for these policies, the number of additional misses is not bounded by a constant.

3.2. Data-flow analysis

To derive the set of DC-UCBs, we present a data-flow analysis that builds upon the results of a general cache analysis as it is typically used for WCET analysis. Such a cache analysis only must be able to safely classify accesses into hits, misses or unknown. While UCB analysis of instruction caches is derived at a basic block level, all DC-UCB analyses (data and instruction caches) are derived at an instruction level due to the incorporation of the results of the cache analysis; the cache analysis delivers the information, "is cached or not" per instruction. Therefore, for DC-UCB analysis, the following notations are used:

We denote instruction j of basic block i with B_i^j. We use the partial function $Access_D(B_i^j)$ to denote the memory block of a possible data access during execution of instruction B_i^j. In case no data memory is accessed, $Access_D(B_i^j)$ is not defined. Furthermore, we use $Access_I(B_i^j)$ as the address of the instruction. Note that we omit the index D, I in case no distinctions need to be made or when it is determined by the context.

3.2.1. DC-UCB analysis

Remember that a memory block m is called a definitely-cached UCB at program point P if (a) m *must* be cached before the preemption point and (b) m *may* be reused at program point Q that *may* be reached from P and *must* be cached along the path to its reuse. As we take into account the cache analysis derived as part of the WCET analysis, only one data-flow analysis—to check for condition (b)—is needed to derive the set of DC-UCBs: This data-flow analysis uses the result of the must analysis at each program point, which checks for condition (a).

To determine the set of definitely-cached UCBs, we use a backward program analysis on the control flow graph. A memory block m is added to the set of DC-UCBs of instruction B_i^j, if m is element of the must cache at B_i^j and if instruction B_i^j accesses m. The domain of our analysis is the powerset domain on the set of memory blocks M: $\mathbb{D} = 2^M$. The following two equations determine the flow information before (DC-UCB_{in}) and after (DC-UCB_{out}) instruction B_i^j:

$$\text{DC-UCB}_{in}(B_i^j) = gen(B_i^j) \cup (\text{DC-UCB}_{out}(B_i^j) \setminus kill(B_i^j)) \tag{9}$$

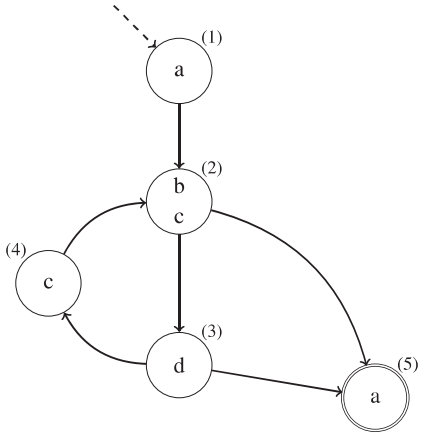
$$\text{DC-UCB}_{out}(B_i^j) = \bigcup_{\text{successor } B_k^l} \text{DC-UCB}_{in}(B_k^l) \tag{10}$$

where the gen/kill sets are defined as follows:

$$gen(B_i^j) = \begin{cases} \{Access(B_i^j)\} & \text{if } Access(B_i^j) \in Must\_Cache(B_i^j) \\ \emptyset & \text{otherwise} \end{cases} \tag{11}$$

$$kill(B_i^j) = M \setminus Must\_Cache(B_i^j) \tag{12}$$

The direction of the analysis is backward. Eq. (10) combines the flow information of all successors of instruction B_i^j. Eq. (9) represents the update of the flow information due to the execution of the instruction. First, all memory blocks not contained in the must



B_i^j	Must-Cache	DC-UCBs$_{in}$	UCBs$_{in}$
B_1^1	$(\top, \top, \top, \top)$	$(\top, \top, \top, \top)$	$(\top, \top, \top, \top)$
B_2^1	$(a, \top, \top, \top)$	$(a, \top, \top, \top)$	(a, b, c, d)
B_2^2	$(a, b, \top, \top)$	$(a, \top, \top, \top)$	(a, b, c, d)
B_3^1	$(a, b, c, \top)$	$(a, \top, c, \top)$	(a, b, c, d)
B_4^1	(a, b, c, d)	$(a, \top, c, \top)$	(a, b, c, d)
B_5^1	$(a, \top, c, \top)$	$(a, \top, \top, \top)$	(a, b, c, d)

Fig. 8. Example control flow graph and corresponding must cache, DC-UCBs and UCBs.

Table 2
Equations of DC-UCB analysis for CFG of Fig. 8.

B_i^j	DC-UCB$_{in}(B_i^j)$	DC-UCB$_{out}(B_i^j)$
B_1^1	DC-UCB$_{out}(B_1^1) \cap \emptyset$	DC-UCB$_{in}(B_1^1)$
B_2^1	DC-UCB$_{out}(B_2^1) \cap \{a\}$	DC-UCB$_{in}(B_2^2)$
B_2^2	DC-UCB$_{out}(B_2^2) \cap \{a,b\}$	DC-UCB$_{in}(B_3^1) \cup$ DC-UCB$_{in}(B_5^1)$
B_3^1	DC-UCB$_{out}(B_3^1) \cap \{a,b,c\}$	DC-UCB$_{in}(B_4^1) \cup$ DC-UCB$_{in}(B_5^1)$
B_4^1	$\{c\} \cup$ (DC-UCB$_{out}(B_4^1) \cap \{a,b,c,d\}$)	DC-UCB$_{in}(B_2^1)$
B_5^1	$\{a\} \cup$ (DC-UCB$_{out}(B_5^1) \cap \{a,c\}$)	$\emptyset$

cache at B_i^j are removed from the set of DC-UCBs (12)—only a memory block that is element of the must cache all along the way to its reuse is considered useful by our definition. Then, the accessed memory block of instruction B_i^j is added in case it is contained in the must cache at the instruction (11).

Using these equations, the set of UCBs can be computed via fixed-point iteration [12]. The initial values at instruction B_i^j are defined as follows: DC-UCB$_{in}(B_i^j) = gen(B_i^j)$ and DC-UCB$_{out}(B_i^j) = \emptyset$. Note that Eq. (9) obeys a distributive structure to ensure that the fixed-point iteration will derive the smallest solution to the set of equations. In terms of program analyses, this means that the *minimal fixed-point* solution (MFP) equals the *meet-over-all-paths* solution (MOP). Note that in the later discussion, we use a simplified version of Eq. (9):

$$\text{DC-UCB}_{in}(B_i^j) = gen(B_i^j) \cup (\text{DC-UCB}_{out}(B_i^j) \cap Must\_Cache(B_i^j)) \quad (13)$$

Consider the control flow graph given in Fig. 8. The numbers in the upper-right corner of the boxes denote the different basic blocks. The letters within basic blocks denote the memory blocks accessed by the instructions. The table shows the content of the must cache assuming a direct-mapped data cache of size 4 and the obtained DC-UCBs. Table 2 lists the equations our analysis uses and Table 3 the steps and the resulting sets of the fixed-point iteration. Note that we use the simplified Eq. (13). Memory block a is cached before and reused at instruction B_5^1. So a is a DC-UCB before B_5^1 and contained in the initial state of DC-UCB$_{in}(B_5^1)$ (Table 3, $it = 0$). The same holds for memory block c and instruction B_4^1. In the next step of the computation ($it = 1$), a is furthermore considered a DC-UCB at B_5^1's predecessors B_2^2 and B_3^1, and c at B_4^1's predecessor B_3^1. The equations provided in Table 2 are iteratively applied until a fixed-point on the set of DC-UCBs is reached. Note that all prior UCB analyses employ two program analyses, one to check for condition (a) memory block may be cached at position P, the other for condition (b) memory block may be reused. If we adhere to this structure, we can see the cache analysis as presented in the Section 1.2 as one program analysis (a)—the analysis which we presented above as the other one (b).

3.2.2. LRU caches

Although the DC-UCB analysis was introduced using a direct-mapped cache, the analysis is valid in the same manner for set-associative or fully-associative LRU caches. The DC-UCB analysis only relies on the information about the currently accessed memory blocks and on an under-approximation of the cache content, i.e., the must cache. It simply restricts the live memory blocks to blocks contained in the must cache. The structure of the cache (as well as the replacement policy and cache-set mapping) is completely masked by the set $Must\_Cache(B_i^j)$, which is obtained by the cache analysis as part of the WCET analysis. Therefore, if the WCET analysis is able to handle LRU caches, so is the DC-UCB analysis. In contrast the initial UCB analysis for set-associative caches, where each cache set is modeled explicitly, all DC-UCBs are kept in one global DC-UCB set.

3.2.3. Data caches/dynamic memory accesses

If the addresses of the memory accesses are known statically, the same DC-UCB analysis as for instruction caches can be applied. For some data accesses, however, the value analysis [3] is unable to derive precise addresses statically. Consider an array access within a loop, for instance. The actual address of the access changes each iteration. The value analysis, thus, derives a range bounding the address of the access instead of a single value. We call such a memory access a dynamic memory access. In addition to $Access(B_i^j)$ which denotes the address of the access, $Range(B_i^j)$ denotes the length of it. In case no range is needed, $Range(B_i^j) = 1$. As for the cache analysis, the DC-UCB analysis has to assume results of a sound and safe value analysis given. The actual implementation does not matter.

To handle imprecise information about the addresses of memory accesses—and to enable a general DC-UCB analysis for data caches[2]—we have to adapt the analysis as follows. The domain is given by a multiset: $\mathbb{D} = 2^{(M,I)}$ where I denotes the length of the range given in the number of memory blocks. An access whose effective address is bounded by a memory range is only considered a cache hit if the whole range is definitely cached. Therefore, the access is also only considered a definitely-cached UCB in this case (14). Elements are removed from the set of DC-UCBs only if no memory block is definitely cached (15). Otherwise, if one memory block is cached and this memory block is removed due to preemption, it may cause an additional cache miss. The corresponding kill/gen sets are specified as follows:

$$gen(B_i^j) = \begin{cases} \{(Access(B_i^j), Range(B_i^j))\} & \text{if } \forall_{l=0}^{l<Range(B_i^j)}(Access(B_i^j)+l) \\ \in Must\_Cache(B_i^j) & \\ \emptyset & \text{otherwise} \end{cases} \quad (14)$$

$$kill(B_i^j) = \{(Access(B_k^m), Range(B_k^m)) | \forall_{l=0}^{l<Range(B_k^m)}(Access(B_k^m)+l) \notin Must\_Cache(B_i^j)\} \quad (15)$$

An example of how the extension of the analysis works is shown in Fig. 9. Assume that a, b and c are sequentially ordered memory blocks. Since the entire range of memory blocks the last instruction may access are cached, the timing analysis considers the data access of this instruction a cache hit. For the same reason, the definitely-cached UCB analysis adds $(a, 3)$ to the set of DC-UCBs. Only before the execution of the first and after execution of the last instruction, $(a, 3)$ is not a definitely-cached UCB; at every other position, a cache block evicted due to preemption may cause an additional cache miss at the last memory access.

3.2.4. Evaluation

In this section, we evaluate the precision of our approach. Hence, we compare our results with the number of UCBs using the initial approach by Lee et al. [7] to see the improvement solely based on the adapted notion.

The evaluation setting is the following. The target architecture is an Arm7 processor[3] with direct-mapped instruction caches of size (a) 1 kB, line size 8 byte and (b) 8 kB, line size 8 byte. The Arm7 features an instruction size of 4 byte. The test cases are taken from the Mälardalen WCET benchmark suite.[4] Table 4 shows the tasks, the number of instructions of each task, the WCET and ratios of task size to cache size. We compiled these tests using a gcc cross-compiler.

[2] If the value analysis can not even derive a memory range for an access, timing analysis treats this memory access as a cache miss. It therefore does not contribute to the bound on the CRPD.

[3] <http://www.arm.com/products/CPUs/families/ARM7Family.html>.

[4] <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>.

Table 3
Fixed-point iteration to derive the set of DC-UCBs for CFG of Fig. 8.

it	DC-UCB$_{in}$						DC-UCB$_{out}$					
	B_1^1	B_2^1	B_2^2	B_3^1	B_4^1	B_5^1	B_1^1	B_2^1	B_2^2	B_3^1	B_4^1	B_5^1
0	{}	{}	{}	{}	$\{c\}$	$\{a\}$	{}	{}	{}	{}	{}	{}
1	{}	{}	$\{a\}$	$\{a,c\}$	$\{c\}$	$\{a\}$	{}	{}	$\{a\}$	$\{a,c\}$	{}	{}
2	{}	$\{a\}$	$\{a\}$	$\{a,c\}$	$\{c\}$	$\{a\}$	{}	$\{a\}$	$\{a,c\}$	$\{a,c\}$	$\{c\}$	{}
3	{}	$\{a\}$	$\{a\}$	$\{a,c\}$	$\{a,c\}$	$\{a\}$	$\{a\}$	$\{a\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$	{}

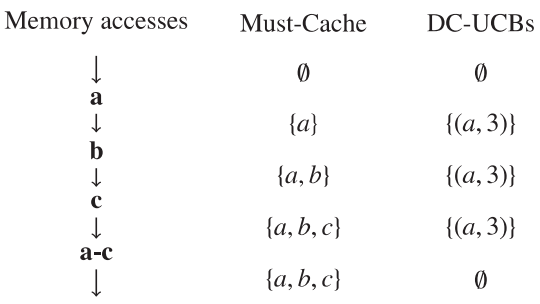


Fig. 9. Data cache DC-UCB analysis handling memory ranges; a sequence of memory accesses, cache content and sets of DC-UCBs.

To tighten the bound on the worst-case execution time, a technique called virtual inlining and loop unrolling, cf. [18], is applied. This technique artificially increases the control flow graph to distinguish between different loop iterations and function calls. It especially improves the must-cache analysis, i.e., more memory accesses will be classified as cache hits. For all test cases, we also employed virtual unrolling and virtual inlining in order to derive realistic results. Note that by improving the precision of the must cache, we increase the sets of DC-UCBs and so, we can only decrease the improvement we obtain.

Fig. 10 shows the results for the different cache sizes. The average size of the (DC-)UCB sets per instruction (columns 2/8 and 3/9) can be seen as an indicator of the CRPD in case preemption is restricted to a given set of preemption points. Column 4/10 show the average improvement, i.e., the sum over the improvements of each instruction divided by the number of instructions. Columns 5/11 and 6/12 show the maximal number of (DC-)UCBs for the given task and the last column the improvement. The maximal number of (DC-)UCBs multiplied by the cache reload cost constitutes an overall upper bound on the CRPD of the whole task (when combined with an upper bound of the WCET). Columns 7/13 show the relative improvement of our new approach, i.e., $(|\text{UCB}| - |\text{DC-UCB}|)/|\text{UCB}|$.

The overall impact of the improvement to the schedulability of a task set is more complex to determine. Of course, one could simply compare the CRPD to the WCET. These results, however, are not very useful: (a) the WCET strongly depends on the loop bounds, whereas the CRPD is nearly independent of them and (b) a task with a high WCET is likely to be preempted several times, whereas a task with a short WCET is probably not preempted at all. Hence, we assume a simple schedule in the following way: each of the tasks of Table 4 is scheduled with an artificial higher-priority task with a period of 10,000 cycles. So, each task is preempted up to $\lceil \text{WCET}/10,000 \rceil$ times. We restrict these comparisons to the first cache configuration (8 kB, line size 8 byte) and a cache reload time of four cycles. Also the WCETs are derived for such a cache using the timing analysis tool *aiT*.[5] The results are shown in Fig. 11. The WCET of each task is set to 100%. The two bars show CRPD in relation to the WCET, the first one based on the CRPD computation using UCBs, the second one using DC-UCBs.

[5] <http://www.absint.com>.

3.2.4.1. Discussion. The results show that our analysis (DC-UCB) strongly outperforms the former approach (UCB). Depending on the structure and the size of the task, the improvement of the maximal number of UCBs ranges from at least 18% (*minmax*) up to 97% (*qurt*, cache size = 4 kB, 8 kB). In case the cache is small compared to the task size (see Table 4), the number of UCBs is often bounded by the number of cache sets (cache size 1 kB, *qurt, select, sqrt*). Thus, the difference to the number of definitely-cached UCBs is smaller and improvements are less obvious.

Task *minmax* contains no loop and the useful cache blocks occur in a function invoked twice. Therefore, the number of (DC-)UCBs in both analyses is rather small and both approaches differ only slightly. Also for *loop3*, DC-UCB only slightly improves the results. Again, the number of UCBs is already small, this time because the task contains a large sequence of very small non-nested loops. Best improvements are observed for programs that containing large loops (sqrt, qurt, crc), recursive structures, or repeated invocation of routines. In these cases, the cache can work quite effectively which means that several cache blocks will be reused and memory accesses result in cache hits. Lee et al. over-approximates—in his notion of UCBs—the sets of these memory accesses. However, since the must cache can only classify a subset of them as cache hits, only a strongly reduced set is considered as definitely-cached UCBs. Of course, the improvement could even be more impressive, if we would have used a worse must-cache analysis. But due to virtual loop unrolling and virtual inlining, we used a very precise must-cache analysis to derive realistic results.

A cache block typically contains more than one instruction. So, even for straight-line code sequences without loops, the sets of UCBs and DC-UCBs are not completely empty. Since all programs contain such fragments with instructions executed at most once, the average number of (DC-)UCBs per instruction[6] is reduced. Thus, the average improvement per instruction (Column 4/11, Fig. 10) is always lower than the improvement on the maximal number. Nevertheless, values up to 80% are still possible. This indicates a large refinement of the CRPD also in case preemption is deferred to a fixed set of preemption points.

To sum up, the difference between both notions can be explained due to difference between the sets of possibly-cached and definitely-cached memory blocks. To derive a general safe upper bound on the cache-related preemption delay the set of possibly-cached memory blocks must be considered. However, since the CRPD is always used in combination with the bound on the worst-case execution time, it is sufficient to consider the set of definitely-cached memory blocks. So, the DC-UCB analysis only accounts for cache misses that are not taken into account by the timing analysis.

The overall impact of our improvement can be seen in Fig. 11. In most cases, CRPD_{UCB} is at least 20% of the WCET, whereas only in one case the $\text{CRPD}_{\text{DC-UCB}}$ exceeds this limit. Therefore our improvement may have an immense impact on the overall schedulability test.

[6] Although the UCB analysis operates on the basic block level, the set of UCBs can be mapped again to the instructions.

Table 4
Number of instructions and ratios of task sizes to cache sizes.

Task	bs	bsort100	crc	fac	fibcall	fir	insertsort	loop3	matmul	minmax	ns	qsort-exam	qurt	select	sqrt
#Instructions	69	123	288	48	47	209	81	1633	200	138	127	340	967	302	953
WCET	445	1,567,222	290,782	1252	1351	29,160	6573	13,449	742,585	504	43,319	221,46	214,318	17,088	39,962
Ratio 1 kB	0.27	0.48	1.12	0.19	0.18	0.81	0.31	6.38	0.78	0.54	0.50	1.33	3.78	1.18	3.73
Ratio 4 kB	0.07	0.12	0.28	0.05	0.05	0.2	0.08	1.59	0.2	0.13	0.12	0.33	0.94	0.29	0.93
Ratio 8 kB	0.03	0.06	0.14	0.02	0.02	0.10	0.04	0.80	0.10	0.07	0.06	0.17	0.47	0.15	0.47

	Cache size: 1kB, Line size: 8 Byte						Cache size: 8kB, Line size: 8 Byte					
	average			upper bound			average			upper bound		
	UCB	DC-UCB	diff	UCB	DC-UCB	diff	UCB	DC-UCB	diff	UCB	DC-UCB	diff
bs	13.6	1.4	52%	24	5	79%	18.9	1.9	54%	35	8	77%
bsort100	18.9	1.9	54%	35	8	77%	11.3	1.6	61%	20	5	75%
crc	98.7	2.5	84%	124	14	89%	115.0	2.5	84%	134	14	90%
fac	10.8	1.2	51%	19	4	79%	10.8	1.2	51%	19	4	79%
fibcall	5.1	1.6	41%	12	5	58%	5.1	1.6	41%	12	5	58%
fir	47.2	1.9	58%	79	9	89%	47.8	1.9	58%	79	9	89%
insertsort	7.8	2.1	31%	19	10	47%	7.8	2.1	31%	19	10	47%
loop3	3.7	1.5	39%	6	4	33%	3.7	1.5	39%	6	4	33%
matmult	27.3	5.6	56%	40	23	42%	27.6	5.6	56%	40	23	42%
minmax	1.8	1.1	9%	11	9	18%	1.8	1.1	9%	11	9	18%
ns	12.1	2.1	34%	30	13	57%	12.9	2.4	35%	31	13	58%
qsort-exam	101.7	1.9	78%	128	15	88%	127.1	1.9	78%	160	15	91%
qurt	97.2	1.4	75%	128	14	89%	340.8	1.4	76%	449	14	97%
select	94.6	1.8	72%	128	15	88%	102.0	1.8	73%	138	15	89%
sqrt	72.7	1.2	59%	128	14	89%	204.1	1.2	60%	361	14	96%

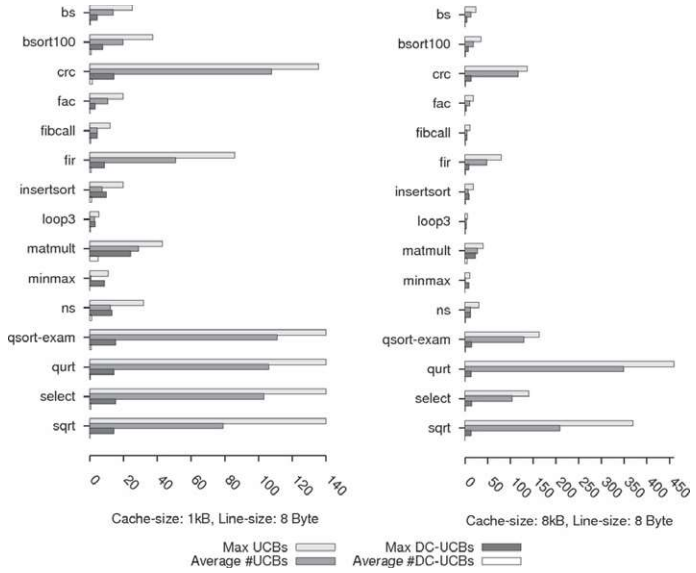


Fig. 10. DC-UCB analysis vs. UCB analysis.

The DC-UCB analysis has the same complexity as the cache analysis; the memory consumption is less since the DC-UCB sets are at most as large as the cache states maintained by the cache analysis. Hence, the run-time of both analyses (Must-Cache and UCB analysis) are very similar.

4. Conclusion

In case of preemptive scheduling, the schedulability analysis has to take into account the context-switch costs for each preemption. These costs are composed of two parts: a constant part depending on the pipeline and on the scheduler invocation and a dynamic part depending on cache interferences. The dynamic part of the context-switch cost is referred to as cache-related preemption delay (CRPD) and represents additional reloads in the cache memory due to preemption.

As presented in the first part of the paper, the CRPD can be bounded by estimating the worst-case effect on the preempted task (useful cache block, UCB) or by estimating the worst-case effect of the preempting task (evicting cache block, ECB). The UCB derivation has been initially introduced for direct-mapped instruction caches, but was soon enhanced to handle set-associative and data caches. As we have shown furthermore, in case of set-associative caches with FIFO/PLRU policies, the CRPD can not be bounded by using these notions; the number of reloads due to preemption can not be bounded by a constant.

The initial UCB definition use an over-approximation of the cache content to derive a safe over-approximation of the sets of UCBs. In combination with the over-approximation of the timing analysis, several cache reloads may be counted twice (as part of the CRPD and as part of the WCET).

So, in the second part of the article, we presented the notion of definitely-cached UCB (DC-UCB). Instead of using an over-approximation of the cache content, our approach uses an under-approximation; we require a useful cache block to be definitely cached. Hence, the number of additional misses due to preemption (not taken into account as misses in the WCET analysis) is given by the

	WCET	CRPD UCB	CRPD DC-UCB	# preemptions
bs	445	96	20	1
bsort100	1567222	21980	5652	157
crc	290782	16320	1680	30
fac	1252	76	16	1
fibcall	1351	48	20	1
fir	29160	948	108	3
insertsort	6573	76	40	1
loop3	13449	192	128	2
matmult	742585	1200	6900	75
minmax	504	44	36	1
ns	43319	620	260	5
qsort-exam	22146	1920	180	3
qurt	214076	39512	1232	22
select	17088	1104	120	2
sqrt	39962	5776	224	4

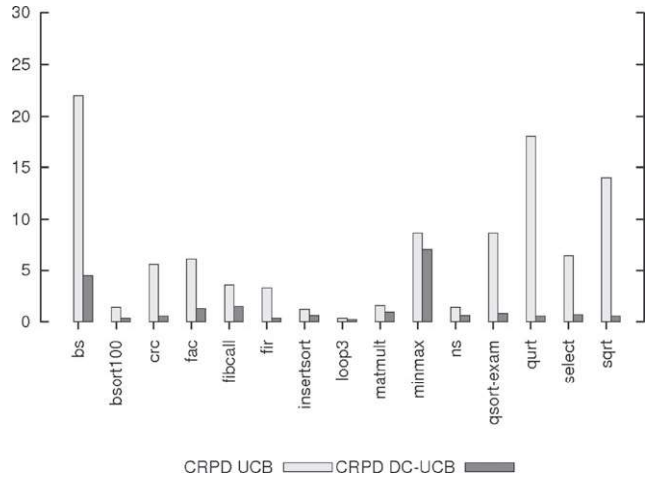


Fig. 11. Overall impact of the DC-UCB analysis to the schedulability test. The *Y*-axis denotes the percentage of the CRPD compared to the WCET.

size of DC-UCBs sets. The data-flow analysis, needed to derive the set of DC-UCBs, uses the must-cache analysis, which is part of the WCET analysis. The comparison of the number of UCBs and DC-UCBs shows the huge improvement on the CRPD bound. As the comparison of WCET and CRPD shows in addition, this refinement has a strong impact on the schedulability analysis for preemptive systems.

As future work, we investigate the improvement by combining DC-UCBs with ECBs. In addition, we plan to compute a lower bound on the CRPD using the notion of useful cache block. Such information may be useful for schedulability analysis, too.

Acknowledgements

We thank Professor Reinhard Wilhelm and Dr. Jan Reineke for support writing this paper. Furthermore, we thank the reviewer for a thorough review and helpful comments.

References

[1] Altmeyer, S., Maiza, C., Reineke, J., 2010. 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. ACM, New York, NY, USA, pp. 153–162.

[2] Burguière, C., Reineke, J., Altmeyer, S., 2009. Cache-related preemption delay computation for set-associative caches: pitfalls and solutions. In: Proceedings of the Workshop on WCET Analysis (WCET'09).

[3] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY, 1977, pp. 238–252.

[4] C. Ferdinand, F. Martin, R. Wilhelm, M. Alt, Cache behavior prediction by abstract interpretation, Science of Computer Programming 35 (2/3) (1999) 163–189.

[5] S. Kim, S.L. Min, R. Ha, Efficient worst case timing analysis of data caching, in: IEEE Real-Time Technology and Applications Symposium (RTAS'96), IEEE, 1996, pp. 230–240.

[6] C.-G. Lee, J. Hahn, S.L. Min, R. Ha, S. Hong, C.Y. Park, M. Lee, C.S. Kim, Analysis of cache-related preemption delay in fixed-priority preemptive scheduling, in: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96), IEEE Computer Society, Washington, DC, USA, 1996, p. 264.

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

[8] S.-S. Lim, Y.H. Bae, G.T. Jang, B.-D. Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park, S.-M. Moon, C.S. Kim, An accurate worst-case timing analysis for RISC processors, IEEE Transactions on Software Engineering 21 (7) (1995) 593–604.

[9] T. Lundqvist, P. Stenström, Timing anomalies in dynamically scheduled microprocessors, in: Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99), IEEE Computer Society, Washington, DC, USA, 1999, p. 12.

[10] F. Mueller, Timing analysis for instruction caches, Real-Time Systems 18 (2000) 209–239.

[11] H.S. Negi, T. Mitra, A. Roychoudhury, Accurate estimation of cache-related preemption delay, in: Proceedings of the 1st ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS'03), ACM, New York, NY, USA, 2003, pp. 201–206.

[12] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[13] H. Ramaprasad, F. Mueller, Bounding preemption delay within data cache reference patterns for real-time tasks, in: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), IEEE Computer Society, Washington, DC, 2006, pp. 71–80.

[14] Reineke, J., 2008. Caches in WCET Analysis. Ph.D. Thesis, Universität des Saarlandes, Saarbrücken.

[15] J. Staschulat, R. Ernst, Scalable precision cache analysis for preemptive scheduling, in: Proceedings of the 2005 ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05), ACM, New York, NY, USA, 2005, pp. 157–165.

[16] J. Staschulat, R. Ernst, Scalable precision cache analysis for real-time software, Transactions on Embedded Computing Systems 6 (4) (2007) 25.

[17] Tan, Y., Mooney, V., 2004. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In: Proceedings of the 8th International Workshop SCOPES 2004. Lecture Notes on Computer Science (LNCS), vol. 3199. Press, pp. 182–199.

[18] Theiling, H., 2002. ILP-based interprocedural path analysis. In: Proceedings of the EMSOFT 2002, Second Workshop on Embedded Software.

[19] H. Tomiyama, N.D. Dutt, Program path analysis to bound cache-related preemption delay in preemptive real-time systems, in: Proceedings of the 8th ACM International Workshop on Hardware/Software Codesign (CODES'00), ACM, New York, NY, USA, 2000, pp. 67–71.

[20] R.T. White, C.A. Healy, D.B. Whalley, F. Mueller, M.G. Harmon, Timing analysis for data caches and set-associative caches, in: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97), IEEE Computer Society, Washington, DC, USA, 1997, p. 192.

Sebastian Altmeyer received his M.Sc. in Computer Science at the Saarland University, Germany in 2006. He is currently a Ph.D. student and research asisstent in the team of Prof. R. Wilhelm, Department of Computer Science, Saarland University, Germany. His research interests include embedded systems, timing and scheduling analysis and abstract interpretation.

Claire Maiza Burguière received her Ph.D. degree in Computer Science from the University of Toulouse, France in 2008. She is currently a postdoc in the team of Prof. R. Wilhelm, Department of Computer Science, Saarland University, Germany. Her research interests include timing analysis, abstract interpretation, cache analysis, WCET and CRPD analysis, architecture of real-time systems and predictability of multi-core architecture.