A Synergetic Approach to Accurate Analysis of Cache-Related Preemption Delay*

Jan C. Kleinsorge
Informatik 12
TU Dortmund
D-44221 Dortmund, Germany
Jan.Kleinsorge@tu-dortmund.de

Heiko Falk
Institute of Embedded Systems/Real-time Systems
Ulm University
D-89081 Ulm, Germany
Heiko.Falk@uni-ulm.de

Peter Marwedel
Informatik 12
TU Dortmund
D-44221 Dortmund, Germany
Peter.Marwedel@tu-dortmund.de

ABSTRACT

The *worst-case execution time* (WCET) of a task denotes the largest possible execution time for all possible inputs and thus, hardware states. For non-preemptive multitask scheduling, techniques for the static estimation of safe upper bounds have been subject to industrial practice for years. For preemptive scheduling however, the isolated analysis of tasks becomes imprecise as interferences among tasks cannot be considered with sufficient precision. For such scenarios, the *cache-related preemption delay* (CRPD) denotes a key metric as it reflects the effects of preemptions on the execution behavior of a single task. Until recently, proposals for CRPD analyses were often limited to direct mapped caches or comparably imprecise for k-way set-associative caches.

In this paper, we propose how the current best techniques for CRPD analysis, which have only been proposed separately and for different aspects of the analysis can be brought together to construct an efficient CRPD analysis with unique properties. Moreover, along the construction, we propose several different enhancements to the methods employed. We also exploit that in a complete approach, analysis steps are synergetic and can be combined into a single analysis pass solving all formerly separate steps at once. In addition, we argue that it is often sufficient to carry out the combined analysis on basic block bounds, which further lowers the overall complexity. The result is a proposal for a fast CRPD analysis of very high accuracy.

*The research leading to these results has received funding from the European Community's ArtistDesign Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'11, October 9–14. 2011, Taipei, Taiwan.
Copyright 2011 ACM 978-1-4503-0714-7/11/10 ...$10.00.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Reliability, Verification, Performance

Keywords

Cache-related Preemption Delay, Useful Cache Blocks, Multitasking, Static Analysis

1. INTRODUCTION

Systems with hard real-time constraints depend on the knowledge of strict timing bounds of their components. In software systems, the *worst-case execution time* (WCET) of one or multiple tasks is one of the key metrics used to reason about the overall timing behavior in such scenarios. A safe bound on the WCET of a single task is typically obtained by means of static analysis. This enables the determination of worst-case system states at micro-architectural level, independent of actual runtime parameters that would otherwise affect the execution time in runtime profiling approaches.

In a non-preemptive multitask system, WCET values of the tasks analyzed as separate, isolated entities are sufficient to reason about the feasibility of a given schedule. That is, whether all tasks meet their deadlines. However, non-preemptiveness severely deteriorates task responsiveness, since high-priority tasks are not allowed to execute immediately, unless no other task is seizing the respective CPU.

Fully preemptive schedules, in contrast, are inherently responsive. Their drawback, however, is that considerable complexity is added to the static analysis of such scenarios to reason about a system's timing behavior and specifically about the feasibility of the chosen scheduling.

Given the fact that a task can potentially be preempted at any given time, the mere knowledge of the isolated WCET is not sufficient to determine whether a task meets its deadline. Potentially, starting times can be delayed and overall execution times can be prolonged.

In addition, each preemption implies immediate context-switch penalties due to – for example – the disruption of pipelined execution, context-saving, the potential execution of scheduling logic, etc. The most important factor, however,

is the eviction of cache contents due to the execution of higher-priority tasks. Loss of cached data due to eviction does not, as in the former case, affect execution performance immediately and practically independently of the system state, but shows its effects upon the next respective memory access at a later time and under conditions that are highly dependent on the memory layout and runtime behavior of the tasks involved.

The *cache-related preemption delay* (CRPD) accounts for the additional delay that is caused by cache evictions incurred from the set of preempting tasks. It has a significant impact on the estimation of preemption costs and is highly dependent on an accurate knowledge of the possible cache contents on preemption. The CRPD is a vital component of the *worst-case response time* [9] (WCRT) analysis on which practically all approaches to schedulabillity analysis are based on. The CRPD and its safe, accurate and efficient estimation is the topic of this paper.

The two primary aspects in CRPD analysis are, on the one hand, how the cache is affected by preemptions by other tasks within a single instance of a task and on the other hand, the effects on the cache between instances of the same task. The predominating concept for the former is that of *useful cache blocks* [6] (UCB) and this is also the domain we restrict our proposal to.

Overall, this paper focuses on the efficient and highly accurate estimation of the CRPD under periodic, fixed-priority scheduling for k-way set-associative instruction caches with LRU replacement policy. We base the work on the most accurate techniques to date, which have only been proposed incompletely or in limited contexts. Specifically, we construct a UCB analysis for set-associative caches from a highly accurate approach which was limited to direct-mapped caches but was shown to be the most accurate approach in its domain [10]. We apply this analysis on the most accurate algorithm for CRPD estimation for set-associative caches [2], which has only been proposed in general terms and independently of an actual UCB analysis. We propose how these can be combined and how synergetic effects can be exploited to save redundant computations. In principle all proposals can be applied to instruction and data caches. But as we will show later, the handling of instructions yields specific optimization potential to vastly reduce the computational complexity even further and we show that this reduction comes at no loss of accuracy. Additionally, we propose a new safe upper bound on the effects of multiple preempting tasks.

Since all components we base our work on have shown to be the most accurate ones in their specific usage domains, and for which we propose further improvements, to the best of the authors' knowledge our synergetic approach is currently the most accurate UCB-bound CRPD analysis published.

In the following Sec. 2, we discuss related work. Sec. 3 addresses the general construction of the proposed analysis and discusses limitations and improvements in existing approaches. We present novel techniques for the reduction of complexities in existing approaches and specify a complete CRPD analysis. In Sec. 4, results of our proposed technique are presented. The closing Sec. 5 concludes the paper and briefly presents ideas for future work.

2. RELATED WORK

The domain of WCRT analyses is, as of today, well understood and techniques of varying complexity and accuracy have been proposed so far. Some of these have matured to the point of industrial strength. For non-preemptive scheduling, the determination of safe bounds on single task WCET is sufficient to reason about the overall behavior of a set of tasks, as no interference by other tasks needs to be taken into account [17].

For fully-preemptive scheduling with fixed priorities, such as Rate Monotonic Scheduling (RMS) [7], a considerable interference between tasks exist, mainly due to the CRPD.

In the original work on WCRT [15], the CRPD has not yet been considered as a factor. Different approaches have been proposed to circumvent the problem of CRPD altogether by establishing optimized memory layouts to prevent tasks from mutually evicting each other [11, 12].

Either way, if no information on the cache is available, a safe but overly pessimistic bound on the CRPD can be obtained by the assumption that all cache contents are being invalidated when being preempted. In a fully preemptive schedule, this effectively draws the cache useless for scenarios with hard real-time constraints.

Static cache analyses [5, 8] enable the consideration of possible (*may*) or guaranteed (*must*) cache contents to tighten these bounds. Two approaches with varying precision and complexity are presented in [5, 10]. Although the latter (so called "state-based" approach) scales exponentially in space and time in its originally proposed form, it offers high accuracy, which is why this is used as the basis for our proposed approach. Moreover, the authors of [13] have proposed a technique to flexibly scale the complexity of the latter analysis at the expense of precision. The general topic of cache analysis and CRPD computation at a broader perspective is discussed in depth in the following Sec. 3.

With the knowledge of cache contents at runtime, tighter bounds on the CRPD can be determined: The interference of the cached elements of a task with its preempting tasks can be computed statically. In [4, 6], the concept of useful cache blocks (UCB) is introduced, which describe blocks, once accessed and thus, cached, that are being reused at a later time within the same task instance. Hence, evictions of UCB by preempting tasks would potentially lead to cache misses after resumption. This leads to a safe estimation of CRPD by only considering the preempted task, independently of the set of preempting ones.

Variations of the basic idea have been proposed in the following publications. Notably, the concept of *evicting cache blocks* (ECB) [16] does not only allow for the consideration of the eviction of all UCB at preemption, but explicitly considers the cache utilization of the preempting tasks. [16] considers ECB only, independently of the actually preempted task. In [10, 13, 14], the CRPD is bounded by considering both UCB and ECB with varying precision. While most proposals limit themselves to direct-mapped caches for the sake of formal simplification, specific problems related to set-associativity have been overlooked or neglected [3] with the consequence of either being unsound or overly pessimistic.

The authors of [1] have shown that the WCET of an isolated task already accounts for certain types of cache misses, which allowed them to bound the WCRT with so called definitely-cached UCB. This results in even tighter bounds but requires full control over both the WCET and CRPD analyses.

Recently, [2] proposed a technique called *resilience analysis*, which classifies useful cache blocks according to their ability

to “survive” preemptions which tightens CRPD estimations to a degree yet to be matched. The foundation of the proposal is still the UCB analysis, which is left unspecified.

In the following sections, we will demonstrate how the very accurate UCB analysis of [10] can be combined with the resilience analysis of [2] to compute highly accurate CRPD estimations. We show that both techniques are synergetic so as to save unnecessary computations and memory consumption.

3. BOUNDING CRPD

In the following, the basic model of our proposed CRPD analysis is presented. In Sec. 3.1, we discuss the construction of a safe and accurate cache analysis for k-way set-associative caches with LRU replacement policy. In Sec. 3.2, we address how UCB and ECB are computed. In Sec. 3.3, we briefly introduce the analysis of cached block resilience and that unique properties of the previously introduced cache analysis can be exploited to combine both separate analyses into one. In Sec. 3.4, we demonstrate how accurate CRPD can be computed from our results. In Sec. 3.5, we propose techniques to significantly reduce the analysis complexity by showing that is is possible to perform our combined analysis on basic block boundaries.

3.1 Accurate Cache Analysis

As sketched in Sec. 1, the key to the determination of an upper bound on the CRPD is a preferentially accurate, but necessarily safe estimation of the state of the cache memories of a system.

We restrict the following discussion to instruction caches. In principle the techniques can be applied to data caches as well, but this complicates the formalism and, as we will show later, the handling of instructions yields specific optimization potential to vastly reduce the complexity of all analyses presented in the rest of this paper.

In the following, we present the predominating concept in CRPD analysis: useful cache blocks (UCB). We present the basics in Sec. 3.1.1 and discuss the construction of the state-based approach for set-associative caches in Sec. 3.1.2.

3.1.1 Fundamentals

Consider a task’s control flow graph where each instruction represents a program point $p \in P$. We refer to a block of memory as being useful, if it may have been loaded into the cache (by accessing the respective memory block) at some program point p_i and may be accessed again at some point p_j, which is reachable along a path in the control flow graph starting at p_i without being evicted (without preemption) from the cache along this path. This is depicted in Fig. 1. The white circles represent accesses to m along a control flow. As can be seen, m is only classified “useful” at those program points from where an access to m is reachable in forward *and* backward direction without being evicted along the path are classified useful. For paths where this does not hold, m is not useful.

Consequently, when a useful block is evicted because the corresponding task has been preempted after the previous access, the next access to it is a cache miss and thus contributes to the CRPD.

A safe but usually overly pessimistic upper bound on the CRPD for a single preemption can be obtained at the program point with the largest set of UCB, since a preemption at such a point would potentially cause the highest number of evictions from the cache and in turn, may cause cache misses at later points along the control flow. In other words, all UCB are considered invalidated when preempted, so the number of possible future cache misses corresponds to the cardinality of the UCB [6].

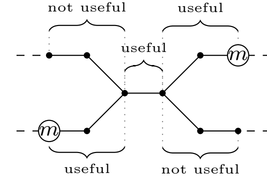

Figure 1: Useful blocks

The contribution of UCB alone to the CRPD can be significantly reduced by obtaining information on the accesses of the preempting tasks, so as to consider only those UCB with actual interference. The interfering blocks of the preempting tasks are termed evicting cache blocks (ECB). As we will see later, ECB are a by-product of the UCB analysis and thus come with negligible extra computational effort.

The precision of the UCB analysis is highly dependent on the cache analysis employed. To statically estimate an upper bound on the UCB in all program points, a so-called *may*-analysis of the cache states is to be performed.

3.1.2 State-based Analysis

Primarily two techniques [6, 10] have been proposed to solve the problem of UCB computation, which differ in precision and complexity and which are both based on finding a fix-point of possible cache memory states over all program points. In the literature, they are known as the “set-based” and “state-based” approaches [13]. In the following, we sketch the construction of the state-based approach which we will use as the basis for our proposal, as we seek to exploit its unique properties as detailed later.

Let c denote the number of *cache sets* and let k denote the cache’s *associativity*. Let $M = \{m_1, m_2, \dots\}$ denote all *memory blocks* mapped to a cache, and let $B = \{b_1, \dots, b_n\}$ denote the set of *cache blocks* where $n = c \times k$ denotes their count.

In k-way set-associative caches, cache blocks are logically grouped into cache sets. Then $set : M \to [1, c]$ maps a memory block to a cache set index.

A cache block uniquely identifies a memory block, if the latter has been loaded. We then write $b_i \mapsto m_j$ to denote this fact.

The approach to UCB analysis as initially proposed in [10] is a highly accurate means to determine cache contents at arbitrary program points. Its purpose is to compute so-called *abstract cache states* which are defined as:

$$\hat{\mathcal{C}} \subset \mathcal{P}(\mathcal{B}_1^k \times \cdots \times \mathcal{B}_c^k) \text{ with } \mathcal{B}_i = M \cup \{\bot\} \quad (1)$$

Where $\hat{\mathcal{C}}$ is the set of all tuples, where each such tuple has $c \times k$ elements representing a memory block or $\bot$, where the latter denotes an undefined mapping. In other words, a single tuple represents one possible state of a cache at a program point. Potentially, multiple of such states may be feasible in a program point. Each $\mathcal{B}^k$ denotes the contents of a cache set. For simplicity, $s_i \in S$ with $|S| = c$ denotes exactly the blocks in cache set i in the following. Note that

cache sets and replacement policy need not be encoded in the abstract state itself as they are purely functional properties.

For example, 3 possible cache states for a cache of $n = 4$ blocks and associativity $k = 1$ could look like:

$$\{(m_1, \bot, m_7, m_4), (m_1, \bot, m_7, m_{12}), (m_9, \bot, m_7, m_4)\}$$

As can be seen, for each cache block with a potentially ambiguous mapping of m a separate cache state exists.

To compute UCB under this model, two iterative data flow equations need to be solved. In a forward-iterative fix-point computation, the *reaching cache states* (RCS) are determined. These represent all feasible mappings of memory blocks onto a cache along the control flow path in every program point $p \in P$.

To this end, we define the generating function which represents the accesses issued in a program point:

$$gen : P \times [1, c] \rightarrow M \cup \{\bot\} \tag{2}$$

with

$$gen(p, i) = \begin{cases} m & \text{if } m \text{ accessed in } p \wedge set(m) = i \\ \bot & \text{otherwise} \end{cases} \tag{3}$$

gen maps to the memory accesses (if any) issued at program point p and which are mapped to cache set i. We assume a single access per cache set at each program point. This is a feasible restriction for the targeted architectures and simplifies the following definitions.

Many set-associative caches depend on the least-recently used replacement policy (LRU) as an eviction policy when a new memory block is loaded into a cache set due to a cache miss. We only sketch the idea.

Let the replacement function $lru : M \times B^k \rightarrow B^k$ for a single cache set (of associativity k) $s = (b_1, \ldots, b_k)$ be defined as:

$$lru(m, s) = \begin{cases} mtf(m, s) & \text{if } m \text{ mapped in } s \\ (m, b_1, \ldots, b_{k-1}) & \text{otherwise} \end{cases} \tag{4}$$

with $mtf : M \times B^k \rightarrow B^k$ defined as:

$$mtf(m, s) = (m, b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_k) \tag{5}$$

When m is not already in the set, lru places it at the head of the queue and therefore drops the oldest element b_k or, if m is already in the set, the function mtf (move-to-front) places it at the head, shifting blocks accordingly.

The data flow equations to compute the RCS for each program point are then defined as:

$$RCS_{in}(p_i) = \bigcup_{p_j \in pred(p_i)} RCS_{out}(p_j) \tag{6}$$

$$RCS_{out}(p_i) = \{t_c(r) | r \in RCS_{in}(p_i)\} \tag{7}$$

with $t_c : s^c \rightarrow s^c$ defined as:

$$\begin{aligned} &t_c((s_1, \ldots, s_c)) = \\ &\quad (t_s(gen(p_i, 1), s_1), \ldots, t_s(gen(p_i, c), s_c) \end{aligned} \tag{8}$$

where $t_s : M \times B^k \rightarrow B^k$ is defined as:

$$t_s(m, s) = \begin{cases} lru(m, s) & \text{if } m \neq \bot \\ s & \text{otherwise} \end{cases} \tag{9}$$

That is, RCS_{in} represents all different preceding abstract cache states and RCS_{out} represents the set of cache states after memory accesses have been applied to sets according to the replacement policy. In Eq. 7, $r \in RCS_{in}$ represents a cache's state, which is just a tuple of cache blocks as defined earlier. For the sake of simplicity, s_i in Eq. 8 represents the blocks composing a cache set. We apply the appropriate memory access for the respective cache set in Eq. 9. When a memory block mapped to the particular cache set which has been accessed (not $\bot$), the replacement policy is applied. Otherwise, the cache set remains unaltered.

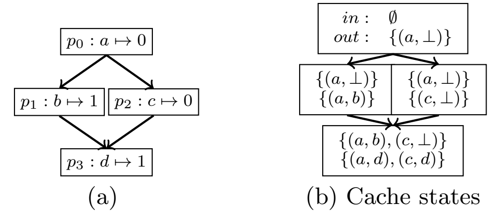

Figure 2: Propagation of state for a direct-mapped cache

The application of the equations is then repeated until a fix-point is reached. For a simple example of the computation of reaching states, consider Fig. 2. We assume a direct-mapped cache for the sake of simplicity. In Fig. 2(a), the mapping of a memory block into a respective cache block for each program point p_i (Eq. 2) is shown. Accordingly, Fig. 2(b) depicts the changes in the *in* and *out* sets due to Eq. 6 and Eq. 7. Note how multiple possible cache states exist after the control flow joins which implicitly encode the different paths taken.

Inversely, the *live cache states* (LCS) can be computed by applying analogous join and update functions during backward iteration by collecting states of successors in the control flow graph instead. The LCS represent the memory blocks at a particular program point which may be accessed again at a later time along the control flow without being evicted.

3.2 Computation of UCB and ECB

With RCS and LCS computed, the UCB at each program point $p \in P$ can be determined. Computing the UCB requires to intersect cache states of RCS and LCS. Formally, the set of UCB at some program point p is defined as:

$$UCB^p = \{r \cap_c l | r \in RCS^p_{out}, l \in LCS^p_{out}\} \tag{10}$$

where operator $\cap_c$ on two abstract cache states is defined as:

$$c \cap_c c' := (s_1 \cap_s s'_1, \ldots, s_c \cap_s s'_c) \tag{11}$$

with

$$s_i \cap_s s'_i := \{b | b \in s_i \wedge b \in s'_i\} \tag{12}$$

That is, the UCB at each program point are a set of tuples that represent the pairwise intersection of RCS and LCS. Each tuple element corresponds to a cache set and contains the intersection of blocks mapped onto the respective set. We say a block m is useful in a program point p, iff

$$m \in \bigcup\nolimits_{u \in UCB^p} u \tag{13}$$

and abbreviate this with $m \in UCB^p$ for simplicity.

Note that information on the cache set ordering is dropped. But we still maintain individual cache sets and all cache states.

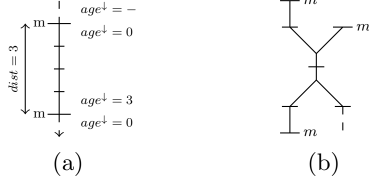

Figure 3: Concept of block age and resilience

In turn, the ECB represent an upper bound on the cache usage of preempting tasks. This enables the computation of much tighter bounds on the task interference, as only those UCB can lead to cache misses that have potentially been evicted by a preempting task, as has been noted in the introduction.

The ECB can directly be derived from the analysis results for the UCB. Similar to the computation of UCB, potentially multiple ECB are feasible for a single program point. Therefore, for a *single* preempting task, we define ECB as the set of all feasible reaching states of the final program point p_{exit}:

$$ECB = RCS_{out}^{p_{exit}} \tag{14}$$

Note that ECB denote which cache sets have been used and to what extent. For RCS specifically, this is more accurate than in other approaches [13] to cache analysis as the set of states only holds those cache states that are actually feasible along all paths leading to the final program point.

3.3 Cached Block Resilience

In the following Sec. 3.3.1, a technique called resilience analysis [2] is sketched, which vastly improves accuracy for UCB-based CRPD for set-associative caches. In Sec. 3.3.2, an alternative algorithm for its computation is proposed and we show that in case of state-based UCB computations, an explicit resilience analysis is redundant.

3.3.1 Concept of Resilience

Given a memory block m is referenced for the first time, it is placed at the head of the logical “queue” of its cache set, which is implied by the replacement policy. An access to a different memory block will increase m's distance to the head position, which we call the block's *age* in the following. As long as $age(m) \leq k-1$, m is not evicted from the cache and therefore causes no cache miss upon its next reference. Obviously, referencing m again resets its age to 0. This is depicted in Fig. 3(a) where the accesses to m and accesses to other memory blocks mapping into the same cache set are marked along the control flow from the first access of m to the next one.

Not only do different accesses from the same task increase m's age, but so do accesses from other tasks, if they happen to preempt the current one after m has been loaded into the cache. Therefore, the accesses of preempting tasks potentially increase the age in addition to the aging happening in the preempted task alone.

Given only the accesses of the preempted task that contribute to m's aging, the maximum amount of *additional accesses* by the preempting tasks without causing a cache miss upon the next access to m is referred to as the block's *resilience* [2]. In the following, we use the term *cached block resiliencies* (CBR).

The CBR are composed of the maximal age from all previous points with an access to m to some program point p (“forward age”) and the maximal aging from p, until the next access to m occurs (“backward age”). Knowing these two metrics allows to reason about whether m is guaranteed to be present in the cache at p at all and whether m remains to be cached even if a preemption at p with known ECB occurs.

Consider Fig. 3(b). Accesses to m are labeled. All other accesses map into the same cache set. The longest path is between the top left to the bottom left access to m. For every point between these two accesses, the sum of the forward and backward ages is constantly 3, even though a shorter path is possible from the top right. On the path to the bottom right from the branch, m is not useful and is therefore ignored in finding the longest path.

Let $age_{in}^{p,\downarrow}(m)$ denote the maximal age from previous accesses of m to just before the current program point p, then we define:

$$age_{in}^{p,\downarrow}(m) = \begin{cases} max_{in}(p,m) & \text{if } m \in UCB^p \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

Where max_{in} denotes the maximal age from the previous access to m to the current point p. max_{in} is subject to further discussion and so the exact definition postponed to Sec. 3.3.2.

If $m \in UCB^p$, then there must be at least one path leading to p on which m is useful. Therefore, on this path, m must be cached and reused before eviction. If m would be evicted due to preemption in p, a later cache miss is possible. This implies that the maximum age is computed only from those paths where m is useful. Otherwise, we default the age to 0. In turn, $age_{out}^{p,\downarrow}(m)$ denotes the ages after changes due to accesses in p have been applied. Analogously, ages in reverse direction ($age^{\uparrow}$) are determined to compute the maximum age to the next access.

From $age^{\downarrow}$ and $age^{\uparrow}$ a distance metric can be computed. By adding both ages, we obtain a value that denotes the length of the longest path between the last and next access, if any existed.

Formally, the maximal distance at a program point p is defined as:

$$dist^p(m) = age^{p,\downarrow}(m) \oplus age^{p,\uparrow}(m) \tag{16}$$

where $\oplus : \mathbb{N}_0^2 \to \mathbb{N}_0$ is defined as:

$$a \oplus a' := \begin{cases} k-1 & \text{if } a + a' \geq k \\ a + a' & \text{otherwise} \end{cases} \tag{17}$$

The value of $dist^p(m)$ denotes the maximal distance between accesses to m among all relevant paths. If m is not useful on *at least one* path through p (Eq. 15), it cannot be in the cache and therefore cannot be evicted later. If the maximal distance exceeds the associativity, then we cannot guarantee resilience on *all* useful paths. The value of $k-1$ is chosen for convenience in that case.

The resilience per memory block m at each program point p can then be defined as:

$$cbr^p(m) = (k-1) - dist^p(m) \tag{18}$$

In case the maximal distance between two accesses is smaller than the associativity k, $cbr^p(m)$ denotes how many additional accesses from preempting tasks are allowed before m is potentially evicted. In the other case, we default to a safe

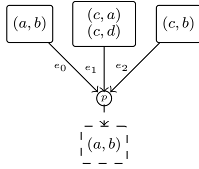

Figure 4: Cache state and block ages

lower bound of 0, which means that we assume that any access from a preempting task evicts m instantly.

3.3.2 CBR from Cache States

In its originally proposed form, the resilience analysis serves as a means to optionally tighten the CRPD. Similarly to ECB, which tighten the bounds on the CRPD estimation by providing information on what blocks *may* be evicted in case of a preemption, the CBR provide information on what blocks *must* be irrelevant to consider as they are so resilient that they can never be evicted.

Contrary to the computation of CBR in its originally proposed form [2], we can exploit the state-based encoding as presented in Sec. 3.1.2 to avoid computing UCB and CBR in separate steps. Nonetheless, we can obtain bounds at least as tight as proposed originally. The separation was necessary mainly due to the fact that the UCB analysis was left unspecified. And arguably, one of the two primary approaches to UCB analysis ("set-based approach" [6]) relies on an abstract state encoding [5] which trades accuracy for memory requirements [13] at a scale, which necessitates the explicit computation of CBR.

Consider Fig. 4, which exemplifies a scenario for a single program point with in-edges e_0 to e_2. The blocks on top denote the possible cache states after the predecessors have been processed according to Sec. 3.2. We assume that each cache state denotes a single cache set, where blocks are aged from left to right. In other words, these states denote the RCS_{out} of the predecessors. The block at the bottom denotes the LCS_{out}, of some successor. That is, blocks that may be reached from our program point without being evicted.

For UCB in p, this means a is useful due to e_0 or e_1 and b is useful due to e_0 or e_2. The RCS_{in} at p is the join of all the states from the predecessors. Therefore, if a is available in one of the incoming states, it is available in the joined states. In other words, a may be cached before p. To derive a safe *age* for a from these states, we first conclude: If a is not useful, it cannot have been cached or, if cached, it cannot reach the point of reuse without being evicted *without preemption* after p. Therefore, the computation of resilience for non-useful blocks need not be taken into consideration. If a is useful in p, it must be among at least one incoming cache state. Since the age of a only matters for states where m actually must have been cached, its maximum age is therefore only to be determined from the states where a is present and iff a is useful. In other words, $max_{in}(p, m)$ from Eq. 15, must be defined as:

$$max_{in}(p, m) = \max_{RCS_{in}(p)} \left\{ \begin{array}{ll} i & \text{if } b_i \in s, b_i \mapsto m \\ 0 & \text{otherwise} \end{array} \right\} \quad (19)$$

where $s = (b_1, \ldots, b_k)$ is the set m is mapped to. That is, we seek the maximum of all ages in the cache sets where m is mapped. If m is not present in some state in RCS_{in}, we assume a safe default age of 0.

To summarize, we exploit the fact that for the state-based encoding, minimum and maximum ages are always exact and all paths leading to different states are always implicitly distinguishable from the states.

3.4 Cache-Related Preemption Delay

In the following, we propose the computation of CRPD. In Sec. 3.4.1, we address the issue of computing safe bounds on combined ECB for multiple preemptions of a task for set-associative caches and present an optimized variant of a previously proposed algorithm which greatly improves its precision. In Sec. 3.4.2, we formulate how the CRPD is actually computed from all that has been discussed so far.

3.4.1 Task Interaction and Safe ECB: A Revision

A critical aspect of the determination of CRPD has not been investigated yet. Although we already introduced ECB, we have not yet addressed their implications on the CRPD computation.

As stated in Sec. 3.2, the ECB for a single preempting task is equal to RCS_{out}. So, ECB are in fact a set of cache states. To simplify matters, we restrict the following discussion to a single element $ecb \in ECB$. Then ecb_i is a state from task T_i. The application to all states should be obvious.

If preemptions nest, that is, they preempt each other in one point of a preempted task, their combined effect on the preempter is basically the set-wise join of all cache sets $s \in ecb$ of all preempting tasks.

If a task is preempted multiple times by multiple tasks, so called "interaction" [2] occurs that might cause more evictions than the separate tasks' *ecb* would suggest. The authors define that two preemptions interact, if there are two consecutive accesses to a memory block m that enclose the preemptions.

For preemptions from different tasks and with an associativity $k > 1$, interaction can cause more evictions than one preemption would cause in isolation.

Consider Fig. 5(a). The circles denote whether m is cached. The cardinality of the ecb_0 and ecb_1 denote the cache set usage of the preempting tasks T_0 and T_1. Without preemption, m does not age from left to right. Therefore, for a resilience of 4, the *ecb* of T_0 alone do not cause an eviction, between two accesses to m. However, if multiple preemptions of different tasks happen to occur between two accesses to m, it is evicted in fact. Therefore, the *ecb* of one task in isolation is insufficient to reason about actual evictions for CRPD computations.

The principle insight is that for each preemption alone m's age only increases by an amount that would not cause an eviction between the two accesses to m. But multiple preemptions in succession possibly increase m's age multiple times before the next access, so that in fact an eviction occurs. Not taking interaction into account underestimates the number of cache misses.

In [2], a safe bound has been proposed for this case. The idea is that we can safely bound interaction by assuming that each preemption by one task always interacts with all other preemptions. We cannot know the actual interaction in a fully preemptive schedule. But if each preemption by a task T_1 is treated as a nested preemption of T_1 with all other tasks of higher priority, then this single preemption causes an aging that is an upper bound on all possible agings through interaction. In other words, the *ecb* for an "artificial"

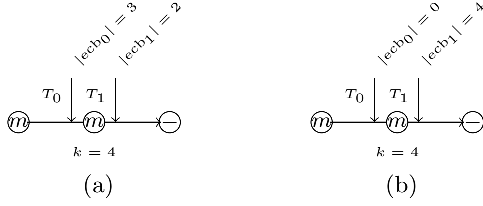

Figure 5: Task interaction

task is computed that denotes a safe bound on the number of evictions one task could cause in interaction with all possible others.

Due to space restrictions, we will directly state their proposed bound here. Assume a set of preempting tasks $\mathcal{T} = (T_1, \ldots, T_n)$, which is ordered by non-increasing numbers of preemptions[1]. Then a safe bound on the CRPD despite interaction is:

$$\mathrm{CRPD}(\mathcal{T}) = \sum_{i=1,\ldots,n} \#P(T_i) \times \mathrm{CRPD}_T(\mathrm{ecb}_i^+) \tag{20}$$

where

$$ecb_i^+ = \bigcup_{j=1..i}^{\leq k} \mathrm{ecb}_j \tag{21}$$

That is, for each preemption of a task T_i, we assume a preemption of an artificial task $T_{1..i}$ instead, that causes a safe amount of aging. The join operator then joins cache set-wise and drops elements if a set's cardinality exceeds k. The total CRPD is then the sum of all per-preempter CRPD weighted by their number of preemptions.

A problem with this bound is that it not only accounts for possible evictions due to interaction. It also unnecessarily accounts for evictions that are caused by the preempters in isolation already, therefore increasing the CRPD by assuming identical evictions multiple times although they are not subject to interaction.

Consider Fig. 5(b). As can be seen, one preemption alone by task T_0 does not contribute to m's aging at all. $T1$ alone then causes the eviction. According to Eq. 21, we assume a miss due to interaction for T_0. When later the per-preemption CRPD for T_1 is computed, the same eviction is counted again.

We propose to only join cache sets in *ecb* conditionally, depending on whether the cache sets in the actually preempting task T_i are non-empty. For an empty set, interaction with other preemptions cannot occur.

Formally, we propose ecb^- instead of ecb^+, which is defined as:

$$ecb^- = \bigcup_{j=1..i}^{\leq k} (ecb_i \otimes_{\leq k} ecb_j) \tag{22}$$

where $e \otimes_{\leq k} e'$ joins all cache sets up to a size of k but only for those whose corresponding sets in the *ecb* of task i is non-empty, since no interaction can occur with regard to such sets.

The extension to all cache states is their cross-product. In the following, when using ECB, we mean ECB^- which is the extension of ecb^- to all cache states whose definition we omit here due to space limitations.

[1]For RMS [7], priority is a monotonically decreasing function of the period.

3.4.2 Computing accurate CRPD

As discussed in Sec. 3.1, the UCB denote an upper bound on the possible number of cache misses, given a preemption at an arbitrary program point. From the previous discussion, we draw the following conclusions:

- The UCB usually vastly overestimate the actual number of evictions: Since with UCB alone, we have no information about which blocks are actually referenced by the preempting tasks and in turn which ones are evicted, the only safe assumption is an eviction of all UCB.
- The amount of additional accesses from a preempting task is already known at this point. ECB denotes an upper bound on their occurrence. That is, the final set of cache states which have previously been determined, indicate to what extent blocks are possibly mapped into cache sets.
- If $cbr^p(m) \geq |ecb_s|$, where $|ecb_s|$ denotes the number of blocks mapped into a set for a single state, then the eviction of m in p is impossible and therefore cannot contribute to the CRPD even though m is useful.

Therefore, a safe bound on the CRPD caused by a single preemption by a single preempting task of a task whose UCB are known, is defined as:

$$\mathrm{CRPD}_T(ECB) = max\{\mathrm{CRPD}^p(ECB) | p \in P\} \tag{23}$$

Over all program points of the preempted task, we take the maximum CRPD which depends on the ECB of the preempters and the UCB in p. To compute CRPD^p, we have to consider all feasible UCB and ECB elements in a point:

$$\mathrm{CRPD}^p(ECB) = \max\{\Delta^p(u,e) | u \in UCB^p, e \in ECB\} \tag{24}$$

Recall that UCB and ECB elements are derived from RCS and LCS cache states. Different states could result in different useful and evicting blocks. So, each combination has to be considered to find the greatest interference. The function $\Delta^p(u,e)$ computes the CRPD over all cache sets and is defined as:

$$\Delta^p(ucb, ecb) = \sum_s \mathrm{CRT} \times \delta_s^p(ucb_s^p, ecb_s) \tag{25}$$

where ucb_s and ecb_s represent useful and evicting cache blocks for single cache sets. CRT denotes the cache reload time. Finally, the CRPD due to a single cache set is defined as:

$$\delta_s^p(ucb_s^p, ecb_s) = |ucb_s^p \backslash \{m | cbr^p(m) \geq |ecb_s|\}| \tag{26}$$

A block which is classified useful and therefore represents a future cache miss can be discarded if its resilience is greater than the amount of disturbance by the preempting task.

3.5 Block-bounded Estimations

In the following, we show that it is safe not to compute states for *all* program points but that we can restrict all analysis steps previously presented to basic block boundaries, significantly reducing the computational effort. Sec. 3.5.1 deals with the general idea for UCB. In Sec. 3.5.2, we show that this is also feasible for CBR at no loss of accuracy, which then allows to fully exploit the combined UCB and CBR analysis as proposed previously.

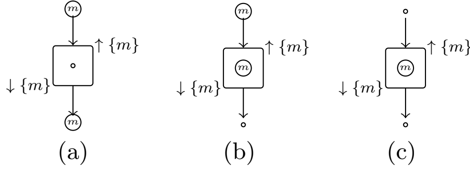

Figure 6: UCB with computations on basic block bounds

3.5.1 UCB on Basic Block Bounds

When performing cache analysis, we relied on the estimation of cache contents at instruction-granularity. This is highly accurate but comes at the price of considerable costs during cache analysis. This manifests in potentially very high execution times and high memory usage. However, we can exploit specific properties of the program structure to significantly lower the amount of relevant program points.

To explain this, recall that a (maximal) basic block bb_i is a maximal sequence of instructions which can only be entered at the first and left after the last instruction, respectively.

It is sufficient to compute UCB for instruction caches at basic block bounds [6]. We restrict the discussion to reaching states. The computation for live cache states should be obvious.

For the states right before and after bb_i, we reuse the terms RCS^i_{in} and RCS^i_{out}, which were previously used for states at instruction-level program points.

When considering all program points within a basic block, the first observation is that only the last k references to memory blocks mapping to the same cache set are relevant for RCS_{out}, since only these will affect the state of the cache set in question. Specifically, m is present in RCS_{out}, iff it has been referenced on the path from the entry of the task to the end of the basic block and is not evicted along that path.

UCB analysis relies on the safe overestimation of block ages. If m is present in RCS_{in} or the last k accesses, then it will be in RCS_{out}. If it is not in RCS_{out}, then it was either in RCS_{in} but is actually evicted within bb_i. Or it was not in RCS_{in} in the first place, which means m has not been referenced at all before, or it has been evicted as it has not been among the last k references of all preceding basic blocks. Analogously, the property holds for the live sets.

Fig. 6 depicts these considerations where $RCS_{out} =\downarrow \{\}$ and $LCS_{out} =\uparrow \{\}$, respectively. In Fig. 6(a), both references are external to the basic block, the memory block (m) being referenced is obviously classified useful. In Fig. 6(b), the block remains useful nonetheless, although m only occurs in the last k accesses. Fig. 6(c) depicts the result, when m is only referenced within this single basic block. In all cases, m remains useful and would therefore safely overestimate the CRPD regardless of where a preemption actually would occur within the basic block.

3.5.2 CBR on Basic Block Bounds

Now that we know that it is possible to significantly reduce the number of program points for which states for UCB computation need to be maintained, it would be most desirable to achieve similar results for CBR. This would allow to compute accurate CRPD with a significant reduction in complexity.

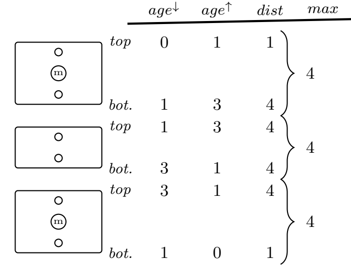

Figure 7: Computing safe block-wide ages

The problem of computing CBR at basic block bounds is that ages must be safe overestimations in each interior point, whereas in the case of UCB, overestimating reachability and liveness at the bounds implicitly overestimates these classifications for the entire basic block. So the question is: Can the estimated age ever happen to be smaller than the real age for all interior points of a basic block?

If we rely only on the last k accesses in a basic block bb, then we can only easily guarantee this property for instruction caches.

Fig. 7 shows how ages at block bounds are computed in principle. An empty circle denotes an access into m's cache set by some other memory block. For each basic block, we need a safe age in forward direction ($age^{\downarrow}$) from the last access to the next access. Analogously, we need this for the backward age ($age^{\uparrow}$).

Given an access to m in some bb, then $age^{\downarrow}_{out}$ denotes the age from the access to the bottom of bb. Inversely, $age^{\uparrow}_{out}$ denotes the aging from the top of bb to the access of m. $age^{\downarrow}_{in}$ and $age^{\uparrow}_{in}$ can be directly computed from RCS_{in} or LCS_{in}.

We only consider forward ages now. Since m is useful in bb, there are two options:

- m has been accessed in some preceding point and is not evicted in all cache states until we reach bb. Then a safe maximal age can be computed from those cache states in RCS_{in} where m is present.
- m is not among any state in RCS_{in}. That means m is useful only because it is accessed within bb for the first time (see Fig. 6). Then $age^{\downarrow}_{in}$ can be ignored and we set its value to 0.

This also directly applies to backward ages and LCS. Note that this is already in accordance to Eq. 15, where the program point is now the age at the top of bb.

Computing the ages at the top and the bottom accordingly, we compute the distance according to Eq. 16, which denotes the maximal path length to and from the access ($dist$ in Fig. 7). The maximum of both distances denotes a safe metric for Eq. 18, where max from Fig. 7 replaces $dist^p$.

If there is no access to m in bb, the aging is obvious. If m is not useful, we skip the computation.

For data caches, the maximum distance between all accesses within the same basic block would have to be taken into account. Notwithstanding, it would be sufficient to keep the maximal ages at the bounds.

4. EVALUATION

In this section, we evaluate the results of our proposed analysis. We conducted the experiments with our WCET-aware C compiler framework (WCC) which was extended

Name	Size	UCB	ECB
adpcm_encoder	2844	59	81
bins	134	5	6
crc	976	17	18
edn	3054	43	98
fft1	4866	56	58
fibcall	56	2	3
jfdctint	2740	52	87
lcdnum	1184	8	10
lms	1834	31	43
matmult	520	9	15
ndes	2586	54	61
qurt	1772	20	21
sqrt	236	7	9
st	1410	20	24

Table 1: Properties of MRTC benchmarks

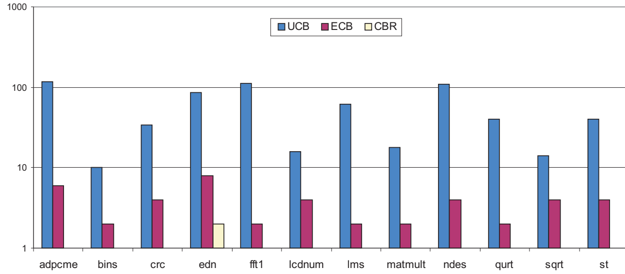

Figure 8: CRPD for a two preemptions by fibcall

by a static cache analysis framework. The latter uses a built-in static pipeline analysis for the TriCore TC1796 CPU to obtain precise static memory access information. The analysis is strictly bound to basic block boundaries.

We chose a 2-way set-associative cache of $4kB$ total size and LRU replacement policy. Although CBR improve the results for increasing set sizes, LRU is rarely applied to caches of more than 4 ways per set.

The evaluation has been carried out on an Intel E5630 (2.53GHz) CPU with no parallel computations.

We made use of the *MRTC* and *PapaBench* benchmark suites to present results on realistic embedded software. We evaluated with floating point operations carried out on the FPU.

For MRTC, we modeled *two* preemptions of a benchmark by a single other benchmark. Due to space limitations, we only present two evaluations with different benchmarks as the preempters. We perform three computations:

- *UCB:* CPRD from only the preemptee's UCB.
- *ECB:* ... from full cache sets that interfere with ECB.
- *CBR:* ... from Eq. 20 with ecb^{-}.

We set the cache reload time (CRT) to 1 cycle, as this only scales the CRPD results but obfuscates the actual effects of UCB, ECB and CBR (see Eq. 25). Note that all diagrams are of logarithmic scale.

For MRTC, we selected a subset of benchmarks to demonstrate the effectiveness of the approach, which are shown in Tab. 1. The size is given in bytes. UCB and ECB are the peek values among all cache states for all program points for the former and those of RCS_{out} for the latter. The analysis time for the benchmarks ranges between 300ms to 14s. On average, 87% less program points were required for the computations when limited to basic block bounds (73 on avg., instead of 328 on avg. at instruction-level granularity).

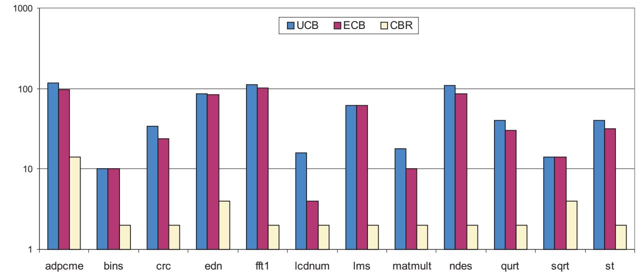

Figure 9: CRPD for two preemptions by jfdctint

Name	Size	UCB	ECB	Period[2]	Priority	Preemptions
T5	166	4	7	375	5	23
T6	752	5	26	375	6	26
T7	164	5	7	75	2	4
T10	462	9	17	375	7	28
T12	700	5	26	75	3	6
I4	338	3	10	150	4	8
I5	260	3	5	75	0	0
I6	108	3	13	75	1	2

Table 2: Properties of PapaBench tasks

In Fig. 8, the CRPD for preemptions by *fibcall* is depicted, which only shows a minimal impact on the preempted tasks due to its small number of ECB. As can be seen, for purely UCB-based computations, the CRPD ranges from 16 misses for *lcdnum* to 118 misses for *adpcm_encoder*. Considering the ECB of the preempter at cache set granularity already causes a reduction of 75% (16 to 4 misses) for the first and 95% (118 to 6 misses) for the latter benchmark. For both benchmarks, the CBR-based analysis reduces the estimated CRPD to 0. Only for *edn* a CRPD of 2 is estimated as opposed to 86 for the purely UCB based computation and 8 for ECB.

Fig. 9 depicts the results for a preempting task with a comparably large number of ECB. The preempting benchmark is *jfdctint*. As can be seen, even though the ECB are taken into account, the CRPD estimation is almost identical to the estimation with UCB alone. This behavior is typical for these estimations when comparably high cache-usage occurs. Particularly in such cases, the CBR-based estimation is superior to the other approaches. For *adpcm_encoder* for example, a 88% (118 to 14 misses) tighter bound is computed, whereas ECB-based estimation is just 8% (118 to 96 misses) tighter than the plain UCB-based computation. The preempters cache usage leads to constantly 2 misses in all preemptees.

PapaBench composes an entire multitask system for an autonomous aircraft. We left out a task (T9) of the benchmark because its disproportional size in combination with its predefined high priority are not beneficial to a meaningful evaluation since it would result in an actual eviction of the entire cache (regardless of what a CRPD analysis would estimate). Execution modes are not distinguished.

Tab. 2 lists the considered tasks' properties. The size is given in bytes, the period is given as a cycle-factor[2]. To obtain deterministic results from identical periods, we manually set fixed priorities by setting periods off by 1 cycle. The priority is 0 for the highest priority task. The last column denotes an upper bound on number of total preemptions of the task. The total analysis time is 7s with only using 2844 computations on basic block bounds as opposed to 18462 computations which would be necessary at instruction-level granularity (85% less).

[2]Period is $value \times 10^8$ cycles (at 150MHz)

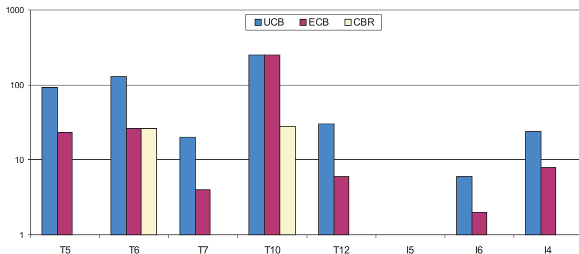

Figure 10: CRPD for all tasks preempted by all higher priority tasks

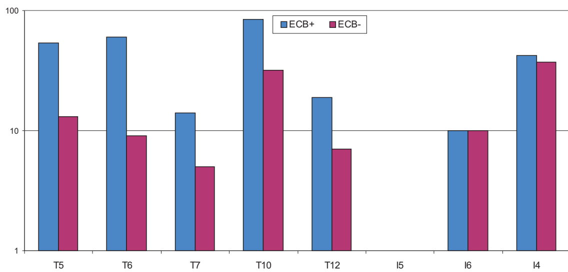

Figure 11: Comparison of ECB^+ and ECB^-

Fig. 10 shows the results of the CRPD computations for all tasks , where each one can be preempted by all higher priority tasks. As can be seen, the UCB-based analysis is significantly overestimating all CRPD except for I5 which is not preempted. Except for T10, considering the ECB at cache set granularity already tightens the CRPD estimation sigificantly (33% for I6). In all cases (except T6, T10), the CBR-based estimation denotes a CRPD of 0. For T6, 80% (130 to 26 misses, ECB and CBR) and for T10 88% (252 to 28 misses, CBR) tighter estimations are computed. In all cases, the CBR-based estimations outperform the UCB-only approach by 89% to 100%.

Fig. 11 depicts the different cardinalities of the peek ECB of the preempters of each task (according to Eqs. 21,22). As can be seen, ECB^+ largely overestimates the ECB in non-trivial cases. The results for I5 and I6, as the two highest priority tasks, are obviously 0 or match the ECB of the one evicting task. For T5, 75% (54 to 13 ECB) tighter bounds are computed with ECB^-. On average we computed 60% tighter bounds.

5. CONCLUSIONS AND FUTURE WORK

We presented how to construct an analysis for CRPD estimation of very high accuracy. To this end, we chose current state-of-the art analysis techniques as the basis for our work, where each one of them has only been presented in a limited context or incompletely.

We showed how a highly accurate UCB analysis which has originally only been proposed for direct-mapped caches can be extended to set-associative caches. As for the resilience analysis, we showed how it can be combined with the proposed UCB analysis to prevent redundant computations. This effectively enables further optimization potential which we exploited in the following.

We also showed that for the computation of ECB for multiple preempting task, a much tighter upper bound can be specified than was previously proposed.

Another aspect is the restricting computations to basic block bounds. Since it is known to be applicable for UCB computations, a synergetic approach necessitates that this is also feasible for CBR. We showed that CBR can be computed on basic block bounds with no loss of accuracy. Therefore, the computational complexity of the combined approach can be significantly lowered.

To the best of our knowledge, a complete approach to CRPD analysis with these components and for set-associative caches has been proposed for the first time. Moreover, since each technique we based our proposal on has been shown to be the most accurate means in their specific contexts, we assume that our proposal is the most accurate purely UCB-based CRPD analysis as of today.

In the future, we intend use our analysis to perform optimizations on schedulings and their parameters, as well as on the tasks themselves to increase the accuracy of estimations and improve optimization potential in hard real-time scenarios.

6. REFERENCES

[1] S. Altmeyer and C. Burguière. A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay. In *Proc. of ECRTS*, 2009.

[2] S. Altmeyer, C. Maiza, et al. Resilience Analysis: Tightening the CRPD bound for set-associative caches. In *Proc. of LCTES*, 2010.

[3] C. Burguière, S. Altmeyer, et al. Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions. In *Proc. of WCET*, 2009.

[4] J. V. Busquets-Mataix, J. J. Serrano, et al. Adding Instruction Cache Effect to Schedulability Analysis of Preemptive RealTime Systems. In *Proc. of RTAS*, 1996.

[5] C. Ferdinand and R. Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Syst.*, 17(2-3), 1999.

[6] C.-G. Lee, J. Hahn, et al. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. In *Proc. of RTSS*, 1996.

[7] J. P. Lehoczky, L. Sha, et al. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proc. of RTSS*, 1989.

[8] Y.-T. S. Li, S. Malik, et al. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proc. of RTSS*, 1996.

[9] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20, January 1973.

[10] H. S. Negi, T. Mitra, et al. Accurate Estimation of Cache-Related Preemption Delay. In *Proc. of CODES+ISSS*, 2003.

[11] S. Plazar, P. Lokuciejewski, et al. WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In *Proc. of WCET*, 2009.

[12] S. Plazar, P. Lokuciejewski, et al. WCET-driven Cache-aware Memory Content Selection. In *Proc. of ISORC*, 2010.

[13] J. Staschulat and R. Ernst. Scalable Precision Cache Analysis for Real-Time Software. *ACM Trans. Embedded Comput. Syst.*, 6(4), 2007.

[14] Y. Tan and V. Mooney. Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-tasking Real-Time Systems. In *Proc. of SCOPES*, 2004.

[15] K. W. Tindell, A. Burns, et al. Extendible Approach For Analysing Fixed Priority Hard Real-Time Tasks. *Real-Time Syst.*, 6, 1994.

[16] H. Tomiyama and N. D. Dutt. Program Path Analysis to Bound Cache-Related Preemption Delay in Preemptive Real-Time Systems. In *Proc. of CODES*, 2000.

[17] R. Wilhelm, J. Engblom, et al. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.