Scalable Precision Cache Analysis for Preemptive Scheduling

Jan Staschulat, Rolf Ernst

Technical University of Braunschweig
Hans Sommer Str. 66, D-38106 Braunschweig, Germany
staschulat|ernst@ida.ing.tu-bs.de

Abstract

Accurate timing analysis is key to efficient embedded system synthesis and integration. Caches are needed to increase the processor performance but they are hard to use because of their complex behavior especially in preemptive scheduling. Current approaches use simplified assumptions or propose exponentially complex analysis algorithms to bound the cache related preemption delay at a context switch. Existing approaches consider only direct mapped caches or propose non conservative approximation for set associative caches.

In this paper we propose a novel cache related preemption delay analysis for set-associative instruction caches where the designer can adjust the analysis precision by scaling the problem complexity. Furthermore, this precise preemption delay analysis is integrated into a scheduling analysis to determine the response time of tasks accurately. In experiments we evaluate this tradeoff between analysis precision and analysis time. The results show an improvement of 22%-71% in analysis precision of cache related preemption delay and 5%-21% in response time analysis compared to previous conservative approaches.

Categories and Subject Descriptors B.3.3 [*Memory Structures*]: Worst-case analysis.

General Terms Algorithms, Measurement, Performance.

Keywords Worst Case Execution Time Analysis, Cache, Embedded Systems, Scheduling.

1. Introduction

Caches are needed to increase processor performance but they are hard to use in real-time systems because of their complex behavior. While it is difficult enough to determine cache behavior for a single task, it becomes even more complicated if preemptive task scheduling is included. Preemptive task scheduling means that task execution can be interrupted by higher priority tasks. In this case, cache improvements can be strongly degraded by frequent replacements of cache blocks.

There are several approaches to make caches more predictable and efficient. One approach is to partition the cache sets and to reserve these partitions for individual tasks. This has been investigated in [13]. The advantage is that cache lines do not have to be reloaded after interrupts and between consecutive executions of the same task. Also, cache behavior becomes (partly) orthogonal for tasks and, therefore, more predictable. Task layout techniques are suggested in [6] which aim at minimizing the inter-task interference in the instruction cache. Another approach is to lock frequently used cache blocks. Such techniques have been investigated by [10] [5] [17] . Both approaches come at an area and power cost as they require a greater caches or memories to become effective. Therefore, heterogeneous memory architectures with caches and scratch-pad SRAM have been introduced [8], where the scratch-pad can hold frequently used cache blocks. Compiler techniques for such architectures have been proposed by [15].

While cache partition and lock strategies are certainly a very useful add-on to improve cache predictability and efficiency, they do not solve the general cache analysis problem which is critical for larger systems of tasks.

Simplified approaches extend the known response time analysis with fixed context switch costs [4] , while more recent approaches [11] [14] use data flow analysis to determine the maximum cache related preemption delay (CRPD) of two tasks. While the approach in [14] is more precise but exponential, the approach in [11] is not precise but polynomial in time complexity.

The first contribution of this paper is a pseudo-polynomial algorithm, where the designer can decide the tradeoff between the analysis precision and analysis execution time. The novel idea is to adjust the level of accuracy by scaling the problem complexity. Secondly, this approach is conservatively extended to set-associative caches. In related work only direct mapped caches are analyzed [14] [16] [21] [4] and the extension to m-way set associative caches in [11] is not conservative, as we will show in this paper. In [12] the CRPD estimation is integrated in a scheduling analysis, but with exponential complexity. Other approaches, such as [16] [4], propose polynomial scheduling algorithms but use simplified assumptions on the CRPD of two tasks. The third contribution of this paper is the integration of the proposed CRPD estimation into a scheduling analysis, such that the entire framework is pseudo-polynomial and is as precise as other polynomial algorithms while some aspects of exponential scheduling algorithms are considered. This approach can be used for timing verification of hard real time systems as well as for design space exploration.

The rest of this paper is organized as follows. Related work is presented in Section 2. In Section 3 the scalable data flow analysis is introduced for direct mapped caches. This modeling is extended in Section 4 to set-associative caches and integrated to a scheduling analysis in Section 5. The results of the experiments are presented in Section 6 before we conclude in Section 7.

LCTES'05 June 15–17, 2005, Chicago, Illinois, USA.

2. Related Work

This section describes related work in the response time analysis for fixed priority preemptive scheduling and reviews current approaches to bound the CRPD.

2.1 Preemptive scheduling analysis

The scheduling method presented in this paper is based on response time analysis (RTA), as described in [4] [3] [9]. The computational model assumes a fixed priority periodic task set with the deadline at the end of a period.

The worst case response time occurs when all tasks are released at the same time point (critical instant). An iterative approach is used to calculate the response time of a given task. The approach tries to allocate in a time window w the task τ_i's computation time C_i, the tasks blocking time B_i and the interference produced by the execution of higher priority tasks. The blocking time is the maximum time that a task can be delayed by lower priority tasks due to resource contention. The process is iterative because in every step the interference is added to the current window w_i^n, resulting in a longer time window w_i^{n+1} that might include greater interference in the next step. The process is finished when the window stops growing ($w_i^{n+1} = w_i^n$). If the resulted response time for any task is greater than its deadline ($w_i^{n+1} = w_i^n = R_i > D_i$), the task-set is not schedulable. The iterative relation is shown below:

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

The term $hp(i)$ denotes the set of tasks with a higher priority than task τ_i.

2.2 Accounting for cache interference

In an embedded system with cache the context switch time depends on the contents of the cache. When a task τ_j preempts τ_i some cache blocks are removed by τ_j and have to be reloaded by τ_i after it resumes. In [4] five possible ways to determine the CRPD are given: 1.) The time to refill the entire cache. 2.) The time to refill the cache blocks displaced by the preempting task. 3.) The time to refill the cache blocks used by the preempted task. 4.) The time to refill the maximum number of useful cache blocks that the preempted task may hold in the cache when a preemption occurs. Useful blocks are those that are likely to be used again [7]. 5.) The time to refill the intersection of blocks between the preempting and preempted task [12].

The approach in [4] considers the penalty according to number 1 and 2 of this list. The following equation is used for the response time analysis. It differs from Equation 1 only by an additional γ_i which corresponds to the additional preemption delay due to the cache interference caused by the preempting task τ_j:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \cdot (C_j + \gamma_j) \qquad (2)$$

In contrast to the work in [4] Petters presents in [16] an approach that considers the preempted task only. The response time of a task is computed by

$$R_i = C_i + B_i + \sum_{j \in hp(i)} (E_j(R_i) \cdot C_j + \Delta_{ji}(R_i)) \qquad (3)$$

where $E_j(R_i)$ denotes the worst case number of releases of task τ_j within the interval R_i. For fixed priority preemptive scheduling this number is given by $E_j(R_i) = \left\lceil \frac{R_i}{T_j} \right\rceil$. To simplify the description, it is assumed that the final response time of a task is already determined ($R_i = w_i^{n+1} = w_i^n$). The maximum preemption delay $\Delta_{ji}(R_i)$ that task τ_j imposes on all tasks with lower priority than τ_j and higher or equal priority then τ_i is computed by an iterative approach. The number of preemptions is bounded by $E_j(R_i)$, that is the maximum number of activations of the preempting task τ_j during the response time of τ_i. Then the $E_j(R_i)$ most expensive preemptions are chosen. The drawback of this approach is that only the useful cache blocks of the preempted task are considered as preemption costs. For larger caches where the preempting and the preempted tasks share only small parts of the cache, this modeling can lead to a pessimistic overestimation.

Approaches such as [12] [14] have been proposed to reduce such overestimation by considering the preempting and preempted task. With a data flow analysis the maximum number of useful cache blocks of the preempted task and the maximum number of used cache blocks of the preempting task are computed. Both sets are intersected to compute the maximum cache related preemption delay. In [12] the response time of a task for fixed priority preemptive scheduling is computed by constructing an integer linear programming problem (ILP). The drawback of their modeling is that an exponential number of equations is necessary to describe the system, making it impracticable for industrial use. Further no exact solution is given, because the worst case preemption cost is assumed for every preemption. This analysis is applied to direct mapped caches and an extension for m-way associative caches is given. However we will show in Section 4.1 that this extension is not conservative. In [14] an approach is proposed for direct mapped caches that analyses the CRPD for each program path based on the approach in [12] but exponential number cache states are necessary in their data flow analysis.

Current approaches that analyze the CRPD are either polynomial but not accurate [12] or exponential because all paths are considered [14]. Current approaches that propose a polynomial response time analysis including CRPD consider either the preempted task only [16] or consider the preempting task only [4]. The approach in [12] which considers the preempted and the preempting task needs an exponential number of equations in the ILP formulation. All approaches restrict the CRPD analysis to direct mapped caches, with one exception. In [11] an non-conservative extension for m-way associative caches is given.

3. Preemption delay analysis

We will motivate our new approach with a small example. A task is represented by its control flow graph (CFG) with basic blocks as nodes and control flow as edges. Figure 1 shows a CFG of a loop with two branches. A node B_i shows the memory blocks of assembly instruction for basic block B_i. For example, the memory blocks m_1, m_2 and m_3 contain assembly instructions that belong to basic block B_2.

We assume a direct mapped cache with 4 cache sets. The mapping of memory blocks to cache sets is given in Figure 1. For example, cache blocks m_0, m_4 and m_8 are mapped to cache set c_0. In Figure 2 the cache contents are shown for each basic block. To abbreviate the notation we use only the index of memory blocks, such that [4 5 - -] represents an entire direct mapped cache with four cache sets with m_4 at cache set c_0, m_5 at c_1 and cache set c_2 and c_3 are empty. A formal definition of a cache state is given in Section 3.2. At basic block B_4 the two incoming cache states from B_2 and B_3 are updated according to the cache mapping in Fig 1, because the memory block m_6 of basic block B_5 is mapped to cache set c_2.

After the second branch at basic block B_7 four cache states would are present. To provide a program-path accurate analysis in the approach in [14] the number of cache states of n consecutive branch statements is given by 2^n, leading to an exponential increase of cache states. On the other side the approach in [11] merges the

contents of every cache set, such that there is at most one cache state. The drawback is that such analysis is not program path-accurate, which means that the cache contents on several paths are merged leading to a reduced analysis precision.

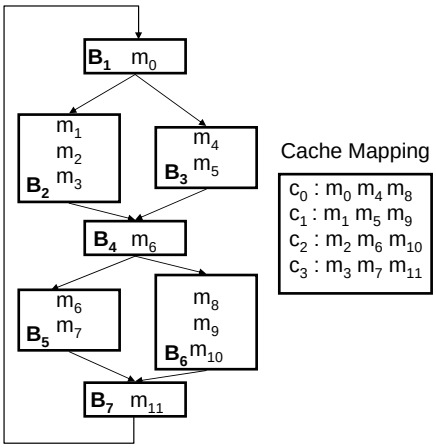

Figure 1. Control flow graph with memory blocks and cache mapping

To reduce the problem complexity we propose to bound the total number of cache states at each node. This way the complexity is reduced by the price of analysis precision. To integrate this idea into CRPD analysis we have to define how to merge two cache states and how to modify the known data flow analysis. For exposition we restrict the presentation to direct mapped caches, which will be extended to set-associative caches in Section 4. We continue the example of Figure 2 in Section 3.5 to compute the cache contents at basic block B_7.

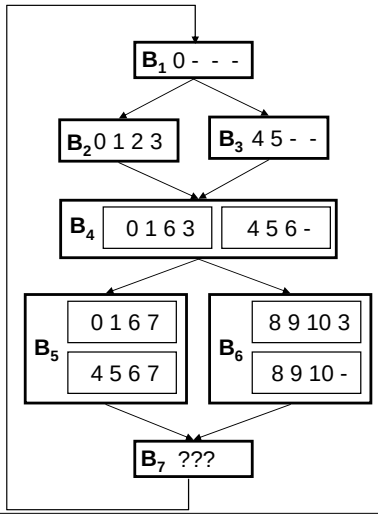

Figure 2. Control flow graph with cache states

3.1 Scope and Limitations

The new approach analyses the preempted task as well as the preempting task to compute the cache related preemption delay for set associative instruction caches. The scheduling analysis computes the response time for a given task set by computing the maximum number of preemptions n and uses the n most expensive preemption costs. During the data flow analysis the designer can determine how many cache states are stored at each node. This allows to scale the problem complexity as well as the analysis precision. The approach assumes the following:

- fixed periodic and fixed priority preemptive scheduling policy, which can be extended to more general activation models (as used in SymTA/S [20]) to model bursts or interrupts.
- constant cache miss penalty to load one cache block from the main memory to the instruction cache
- memory access time is additive to the core execution time. This means that the CPU stalls until the fetched memory block arrives.
- worst case execution time (WCET) is given for each task, including intra-task cache effects.

3.2 General Data Flow Analysis

In a direct mapped cache each memory block can only be mapped into one cache set. In order to merge two cache states we define a cache set as vector of n sets $c[0, \cdots, n-1]$, where $c[i]$ is a *set of memory blocks* and $m \in c[i]$ if cache block i holds memory block m. Otherwise, if the ith cache block does not hold any memory block we denote this as $c[i] = \emptyset$.

We assume that an operation $\odot$ over M (M is the set of all memory blocks) can be applied to a cache state by applying the operation pointwise to its elements. Two cache states c_1 and c_2 can be merged by applying the union operator to each element:

$$c_1 \cup c_2 = (\{c_1[1] \cup c_2[1]\}, \cdots, \{c_1[n] \cup c_2[n]\})$$

In the same way the intersection of two cache states is computed.

This cache state definition differs from the cache state as described in [14] where each $c[i]$ contains exactly one memory block.

A reaching cache state RCS_B at a basic block B of a task is the set of possible cache states when B is reached via any incoming program path. The live cache states at a basic block B, denoted LCS_B, are the possible first memory reference to cache blocks via any outgoing program path from B. RCS_B captures the possible cache states when a task is preempted and LCS_B captures the possible cache usages when the task resumes execution.

As shown in [11] [18] it is sufficient to consider only the end of each basic block as preemption point for CRPD estimation. The sets LCS_B and RCS_B are computed for each basic block B by the fixed-point iterative data flow analysis, as originally described in [1] for optimizing compilers and applied for CRPD estimation in [11] [14] [18].

For the RCS property the quantities RCS_B^{IN} and RCS_B^{OUT} are computed and we set $RCS_B = RCS_B^{OUT}$ if the fixed point is reached. Initially $RCS_B^{IN} = \emptyset$ and $RCS_B^{OUT} = gen_B$. Where gen_B holds all memory blocks introduced into the cache by basic block B. The iterative equations are as follows:

$$RCS_B^{IN} = bound_Z(\bigcup_{p \in pred(B)} RCS_p^{OUT}) \quad (4)$$

$$RCS_B^{OUT} = \{r \odot gen_B | r \in RCS_B^{IN}\} \quad (5)$$

$$c \odot c' = \begin{cases} c' & if \quad c' \neq \emptyset \\ c & otherwise \end{cases} \quad (6)$$

Where c and c' represent a cache set - in both interpretations: cache set as a part of a cache state as well as a mathematical set of memory blocks. The function $bound_Z(C)$ reduces the number of cache states of C to Z elements. This functions allows us to scale

the problem complexity to bound the number of cache states at each node. Its computation is described in Section 3.3

Similarly the LCS property is computed by an iterative fixed point algorithm, the only difference is that the LCS_B^{OUT} is defined by all LCS_B^{IN} sets of all successors of basic block B and the LCS_B^{IN} is computed by applying the $\odot$ operator to the gen_B set and all LCS_B^{OUT} sets. Here the gen_B set is defined by the first memory references in basic block B. Refer to [14] [11] for the details.

3.3 Cache State Reduction

To scale the problem complexity the number of cache sets are bounded. The function $bound_Z(C)$ reduces the set C to Z elements, if $|C| > Z$ otherwise $bound_Z(C) = C$. This is shown in Figure 3. In line 2 the two elements c_i, $c_j \in C$ with the minimum distance $d_{min}(c_i, c_j) = min\{d(c_k, c_l)|c_k, c_l \in C\}$, are chosen. In line 3 these elements are removed from C and in line 4 the merged cache state $c_i \cup c_j$ to C is inserted again. In each iteration the number of elements of C decreases by one, thus this algorithm always terminates.

The distance function $d(c_i, c_j)$ of two cache states c_i, c_j is defined as a metric that delivers the difference of two cache sets. A simple scalar metric d_{scalar} that only counts the number of unequal cache sets, is shown in Equation 7. The index n denotes the total number of cache states.

```
1  while (|C| > Z) do {
2      choose (c_i, c_j) with d(c_i, c_j) minimal
3      C = C \ {c_i} \ {c_j}
4      C = C ∪ {c_i ∪ c_j}
5  }
```

Figure 3. $bound_Z(C)$ algorithm

$$d_{scalar}(c_i, c_j) = \sum_{k=1}^{n} \begin{cases} 1 & if \quad c_i[k] \neq c_j[k] \\ 0 & otherwise \end{cases} \tag{7}$$

A more sophisticated metric, that counts the number of different memory blocks of each cache set with the symmetric difference is given in Equation 8.

$$d(c_i, c_j) = \sum_{k=1}^{n} |(c_i[k] \cup c_j[k]) \setminus (c_i[k] \cap c_j[k])| \tag{8}$$

There are several strategies to choose candidates of cache states of C for merging. One strategy $M1$ is allow only singleton sets, such that the metric is computed between pure cache states (singleton set) and one cache state that might contain (real) sets as its elements. This favors the idea to keep as many separate cache states as possible. The complexity of the merge function $M1$ is proportional to the number of cache states of all predecessor nodes, hence a linear time complexity $O(|C|)$. A second strategy $M2$ applies the metric to all cache states of C. This metric $M2$ requires to compare all pairs of cache states leading to quadratic complexity O($|C|^2$). With same number of elements in C the metric M2 is expected to yield more accurate results than M1, because the cache states with the smallest distance are merged.

3.4 CRPD Computation

The cache related preemption delay is computed by the intersection of useful cache blocks of τ_1 and used cache blocks of τ_2, if τ_2 preempts τ_1. The RCS and LCS properties are computed for each basic block with the iterative data flow analysis described in the previous section. The intersection of both sets RCS_B and LCS_B is the set of useful cache blocks (cache utility vector) CUV_B at basic block B.

The set of used cache blocks is determined by the RCS_{out} of the last basic block of the preempting task τ. As in [14] we define the Final Usage Vector FUV_τ. Suppose that end is the last basic block of the preempting task τ , then $FUV_\tau = \{used(r)|r \in RCS_{end}\}$. The function $used$ is defined over the set of cache states as $used(r) = r'$ where r' is the following bit-vector:

$$r'[i] = \begin{cases} 1 & if \quad r[i] \neq \emptyset \\ 0 & otherwise \end{cases} \tag{9}$$

Finally the CRPD at each basic block B is computed by the intersection of all useful cache blocks CUV_B and used cache blocks of the preempting task FUV_τ.

3.5 Example, continued

We continue the example of Figure 1 and 2. Assume that we want to bound the number of possible cache states to $Z = 2$. At basic block B_7 are four cache states reached via incoming edges. According to the previous sections, we choose the cache states with the minimum distance. The metric d of (8) is used in this example. We denote the cache states of B_5 as $c_1 = (\{m_0\}, \{m_1\}, \{m_6\}, \{m_7\})$ and $c_2 = (\{m_4\}, \{m_5\}, \{m_6\}, \{m_7\})$ and the cache states of B_6 as $c_3 = (\{m_8\}, \{m_9\}, \{m_{10}\}, \{m_3\})$ and $c_3 = (\{m_8\}, \{m_9\}, \{m_{10}\}, \emptyset)$. Then:

$$d(c_1, c_2) = 4 \quad d(c_1, c_3) = 8 \quad d(c_1, c_4) = 7$$
$$d(c_2, c_3) = 8 \quad d(c_2, c_4) = 7 \quad d(c_3, c_4) = 1$$

Therefore we choose to merge $c_3 \cup c_4 = (m_8, m_9, m_{10}, m_3)$ and $c_1 \cup c_2 = (\{m_0, m_4\}, \{m_1, m_5\}, \{m_6\}, \{m_7\}$. These are the missing cache states of basic block B_7 of Figure 2.

Then the computation of RCS is continued as described in equation 5 and 6, until the fixed point is reached. The intersection of RCS and LCS (which are computed similarly) at each node is the set of useful cache blocks which will be intersected with the set of used cache blocks of the preempting task to determine the cache related preemption delay at that basic block.

4. Set associative caches

The general description in Section 3.2 considered only the simplest cache organization: the direct mapped cache. This section extends the proposed analysis to set-associative caches. In a n-way associative cache a memory block can be placed into m cache blocks within its designated cache set. This set-associative cache organization requires a policy called the replacement policy that decides which block to replace when a new memory block is mapped to the cache set and all cache blocks are occupied. The least recently used (LRU) policy, which replaces the block that has not been referenced for the longest time, is typically used for that purpose. In the following, we explain how to compute the number of useful cache blocks for set-associative caches assuming the LRU replacement policy. None of the existing CRPD analysis approaches considered associative caches so far, except in [11]. We show in the Section 4.1 that this modeling is not conservative.

For set associative caches we have to extend the definition of a cache set. A state of a cache set of an n-way set associative cache is defined by a vector $(m_{i_1}, m_{i_2}, \cdots, m_{i_n})$, where m_{i_j} are sets of memory blocks and m_{i_1} is the set that contains the least recently referenced block and m_{i_n} the set that contains the most recently referenced bock.

4.1 Lee et al. approach is not conservative

The approach by Lee et al. [11] extends the direct mapped cache by modeling a cache set as a vector. Because of the limited space

we cannot describe the model in detail, but rather we give an counter example (refer to [11], section 7.1). Assume the cache set vector $c_1 = (m_1, m_2, m_3, m_4)$, where m_1 is the least recently used cache block and m_4 the most recently used cache block. In Lee et al. definition of cache sets are modeled as a vector of memory blocks, not as a vector of sets of memory blocks. Further assume that a basic block B contains the memory blocks m_1 and m_3 which are mapped to c_1. Then similarly the gen set is also represented by a vector: $gen^{c_1} = \{(\bot, \bot, m_1, m_3)\}$ ($\bot$ denotes none memory block). Then the set of reaching memory blocks (RMB) is computed by Equation 10.

$$RMB_{c_1}^{OUT}[B] = \{(m_3, m_4, m_1, m_3)\} \tag{10}$$

Clearly $m_2 \notin RMB_{c_1}^{OUT}[B]$, however the actual cache access results to:

$$\begin{aligned}(m_1, m_2, m_3, m_4)m_1 &= (m_2, m_3, m_4, m_1)\\ (m_2, m_3, m_4, m_1)m_3 &= (m_2, m_4, m_1, m_3)\end{aligned}$$

The elements in the correct cache state (m_2, m_4, m_1, m_3) are only reordered. This case was not considered in Lee et al.'s approach, as equation 10 shows. This leads to a smaller RMB set and to a smaller set of useful cache blocks, which consequently leads to an underestimated cache related preemption delay. We learn from this error, a conservative analysis for set-associative caches has to consider existing elements of the cache state. This is formulated in the next section.

4.2 Data flow analysis

According to our definition in Section 3.2, the RCS_B contains all possible cache blocks at basic block B. In the case of direct-mapped caches, a cache set can hold only one memory block. This modeling has to be extended. In the following we formulate the computation of RCS in data flow analysis terms, we focus on the RCS at the beginning and end of each basic block, as for the case of direct mapped caches. The extension of LCS is analogous.

We define $RCS_c^{IN}[B]$ and $RCS_c^{OUT}[B]$ as the sets of all possible cache states of cache set c at the beginning and the end of basic block B, respectively. The set $gen^c[B]$ contains the state of cache set c generated by basic block B. Its elements have up to n distinct memory blocks that are referenced in basic block B and are mapped to cache set c. More specifically, it is either empty (when none of the memory blocks mapped to cache set c are referenced in B) or a singleton set whose only element is a vector

$$(gen_1^c[B], gen_2^c[B], \cdots, gen_n^c[B])$$

In this vector, $gen_n^c[B]$ contains the memory block which is the last reference to cache set c in B. Similarly $gen_{n-1}^c[B]$ contains the second most recent reference to basic block B and so on. With this definition of $gen^c[B]$, the sets RCS_c^{IN} and RCS_c^{OUT}, whose elements are now vectors of n sets of memory blocks, are related as follows:

$$\begin{aligned}RCS_c^{IN}[B] &= bound_Z(\bigcup_{p \in pred(B)} RCS^{OUT}(p))\\ RCS_c^{OUT}[B] &= \bigcup_{r \in RCS_c^{IN}[B]} LRU_{gen_k^c}(LRU_{gen_{k+1}^c}(\\ &\quad \cdots (LRU_{gen_n}(r)) \cdots)\end{aligned}$$

The function $bound_Z(C)$ is the same as in Figure 3. The extended $LRU_m(c)$ algorithm computes the usual LRU replacement but also considers the case where m is an element of cache set c.

The data flow analysis considers each cache set separately. This is similar to Lee et al. Note, that Z does not bound the number of cache blocks within a set, but rather the total number of cache states, which can contain several (mathematical) sets of cache sets. For example in Fig. 5, basic block B_7 contains two cache states. The cache state $cs_1 \cup cs_3$ again contain two cache sets c_0 and c_1. Cache set c_0 is a set of memory blocks, but there could be more than two elements at each position, independent of the value of Z. The concept for set-associative caches is the same as for direct mapped caches: the parameter Z bounds the number of cache states. It is different in the sense that each cache set itself can contain several sets of memory blocks.

```
Input: cache set c, memory block m Output: cache set c'
0 function LRU_m(c)
1 Initialization ∀i.  c'[i] = ∅
2 if(m ∉ c)
3     c'[n] = {m}
4     c'[j] = c[j+1]   ∀j.  n > j ≥ 1
5 else
6     c'[n] = {m}
7     ∀i.m ∈ c[i] do
8       c'[j] = c'[j] ∪ c[j+1]   ∀j.  n > j ≥ i
9       c'[j] = c'[j] ∪ c[j]      ∀j.  i > j ≥ 1
10      if ( ∃m' ≠ m.  m' ∈ c[i])
11        c'[j] = c'[j] ∪ c[j+1]   ∀j.  n > j ≥ 1
12 remove memory block m from all c'[j].  ∀n > j ≥ 1
```

Figure 4. Extended LRU algorithm

4.3 Extended LRU Algorithm

The function $LRU_m(c)$ implements the LRU replacement strategy for caches states where cache sets are vectors of sets of memory blocks. The algorithm is presented in Figure 4.

Lemma. The algorithm in Figure 4 computes the $LRU_m(c)$ replacement strategy, when memory block m is mapped to cache set c. Provided that c is a vector of sets: $c = [c_1, c_2, \cdots, c_n]$, $c_i \subset M$, where M is the set of all memory blocks and c_1 denotes the least recently used and c_n the most recently used cache block of cache set c.

Proof. The proof is presented over the structure of the cache set elements. We start with the restriction that all c_i contain only one element ($|c_i| \leq 1$) and extend this modeling step-wise to $c_i \subset M$.

Part I. We assume $\forall c_i. \quad |c_i| \leq 1$.

This case represents an ordinary cache state, with n sets for a n way set-associative cache. We distinguish if there exists c_i with $m \in c_i$ or not.

(a) $\forall c_i. \quad m \notin c_i$. The cache block in c_1 will be replaced and the elements will be reordered, such that $c' = [c_2, \cdots, c_n, \{m\}]$. This is implemented in lines 2-4 in Figure 4.

(b) $\exists c_i. \quad m \in c_i$. From assumption $|c_i| \leq 1$ follows that c_i is unique and the loop in line 7 will be executed exactly once. The memory block $m \in c_i$ is placed at the most recently used position c_n and all $c_{i+1} \cdots c_n$ elements shifted one position to the left (lines 6-8): $(c_1, \cdots, c_{i-1}, c_i, c_{i+1}, \cdots, c_n)m = (c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n, \{m\})$.

Note, that the contents of the cache does not change. All elements left from c_i do not change their position (line 9). The condition in line 10 will always evaluate to false, because $|c_i| \leq 1$. Finally the memory block m is removed from the positions 1 till $n-1$ of the cache set vector (line 12). Thus, we have shown that if all $|c_i| \leq 1$ the $LRU_m(c)$ is correct.

Part II. Assumption $\forall c_i. \quad m \notin c_i : |c_i| \leq 1$.

All c_i that do not contain m are singleton sets, only those c_i with $m \in c_i$ may contain more elements.

(a) $m \notin c_i \forall c_i$ this has been shown in (I.)

(b) there exists a unique c_i. $m \in c_i$. If $|c_i| \leq 1$, refer to I, otherwise the case $|c_i| = d \geq 1$ is detected in lines 10-11 in the algorithm. We have to distinguish two cases:

(b1) $(c_1, \cdots, c_{i-1}, \{m\}, c_{i+1}, \cdots, c_n)$

(b2) $(c_1, \cdots, c_{i-1}, \{m_k\}, c_{i+1}, \cdots, c_n), \forall m_k \in c_i . m_k \neq m$

For (b1) we have shown already in I. that $LRU_m(c)$ is correct. In the case of (b2) there are $d-1$ possible cache states, where $m \notin c$. This means that the least recently used memory block c_1 is replaced, the contents of $c_i, i = 2, \cdots, n$ move one position to the left, and $c_n = \{m\}$.

(c) There exist several $c_i . m \in c_i$. Note that in the set representation there may be several sets that contain m, but there cannot be a original cache state with $m \in c_i, m \in c_j, i \neq j$. Thus we can apply lines 8-11 to each cache set c_i that contains m separately and take the union of the resulting cache set c'. Consider we apply the algorithm to some c_i, and there exist $c_{j_1}, \cdots, c_{j_k}$ other sets that contain m. We can formally construct the set of all possible cache states that are described by this cache set and apply the LRU strategy to each cache state, as in part I, and take the union of the resulting cache states. This is implemented in lines 8-9.

Part III. All c_i may have more then one element.

In Part II we have shown that the $LRU_m(c)$ algorithm is correct when all c_i that do not contain m are singleton sets. Part II can be naturally extended to c_i as sets by constructing every possible cache state from the set-based representation, applying the LRU strategy and taking the union of the results. This completes the proof.

4.4 Example

We apply the algorithm to the control flow graph of Figure 1 with an 4-way associative instruction cache with 2 cache sets. For demonstration we compute the reaching cache states (RCS) for each node. Figure 5 shows the same control flow graph with the possible cache states. A cache state consists of two cache sets c_0 and c_1 with each four positions. For example, in basic block B_3 memory block m_0 and m_4 are mapped to cache set c_0 and memory block m_5 to cache set c_1. To save space, m_i is abbreviated as i. An empty position is denoted with $-$. In order to demonstrate the cache state reduction, we restrict the number or cache states to $Z = 2$. Consequently, in B_4 two cache states are reached. In B_4 only the memory block m_6 is mapped to c_0. Since $m_6 \notin c_0$ in both cache states the condition in line 2 is true and the lines 3-4 in the extended LRU algorithm in Figure 4 are executed, which means all memory blocks move one position to the left and m_6 is at the most recently used position. Cache set c_1 is not modified. These two cache states are propagated to B_5 and B_6, where m_6, m_7 and m_8, m_9, m_{10} are accessed respectively.

When the algorithm computes the $RCS^{IN}_{B_7}$ is detects that four cache states $cs_1, cs_2\ cs_3, cs_4$ are available on incoming edges, but only two are allowed. Therefore two of the cache states with the minimum distance according to equation 8 will be merged:

$$d(cs_1, cs_2) = 5 \quad d(cs_1, cs_3) = 5 \quad d(cs_2, cs_3) = 10$$
$$d(cs_1, cs_4) = 10 \quad d(cs_2, cs_4) = 5 \quad d(cs_3, cs_4) = 5$$

We choose to merge $cs_1 \cup cs_3$ and $cs_2 \cup cs_4$.

$$cs''_5 = cs_1 \cup cs_3 = \begin{cases} c_0 & [\{2\}, \{0,6\}, \{2,8\}, \{6,10\}] \\ c_1 & [\emptyset, \{1\}, \{3\}, \{7,9\}] \end{cases}$$

$$cs''_6 = cs_2 \cup cs_4 = \begin{cases} c_0 & [\{4\}, \{0,6\}, \{4,8\}, \{6,10\}] \\ c_1 & [\emptyset, \emptyset, \{5\}, \{7,9\}] \end{cases}$$

In basic block B_7 memory block m_{11} is mapped to c_1. Note that $m_{11} \notin c_1$, such that all elements are only shifted one position to the left by the LRU operator $cs'_5 = LRU_{m_{11}}(cs''_5)$ and $cs'_6 = LRU_{m_{11}}(cs''_6)$. These cache states are shown in basic block B_7.

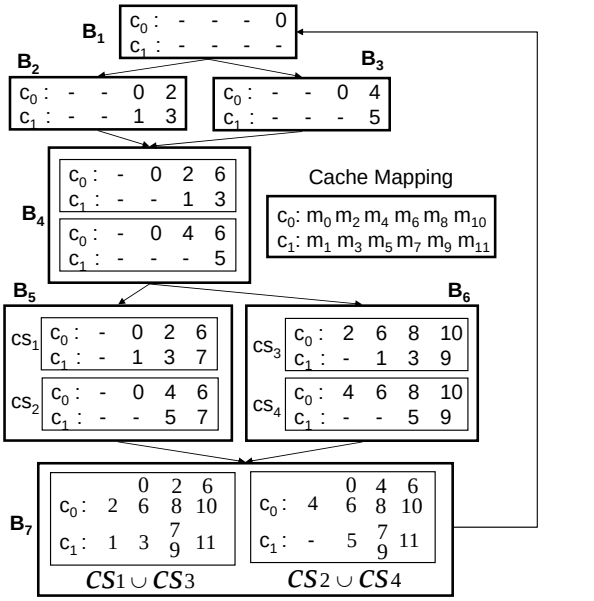

Figure 5. Reaching Cache States for 4-way assoc. cache with 2 sets.

To safe space several elements within a set at a position are aligned vertically, such as m_6 and m_{10} in cache set c_0. For the next basic block B_1 we have a different case, where $m_0 \in c_0$ in both cache states. Therefore the loop in line 7 is executed only once, the contents of $c_0[2]$ till $c_0[4]$ move one position to the left and m_0 is placed in the $c_0[4]$ slot. This results to

$$cs_5 = \text{LRU}_{m_0}(cs'_5) = \begin{cases} c_0 & [\{2,6\}, \{2,8\}, \{6,10\}, \{0\}] \\ c_1 & [\{1\}, \{3\}, \{7,9\}, \{11\}] \end{cases}$$

$$cs_6 = \text{LRU}_{m_0}(cs'_6) = \begin{cases} c_0 & [4, \{0,6\}, \{4,8\}, \{6,10\}] \\ c_1 & [\emptyset, \emptyset, 5, \{7,9\}] \end{cases}$$

The last example of the algorithm is the RCS computation of basic block B_2 with a gen-set=$\{m_1, m_2, m_3\}$ and the cs_5, such that the extended LRU algorithm if Figure 4 is applied three times: $RCS^{OUT}_{B_2} = LRU_{m_3}(LRU_{m_2}(LRU_{m_1}(cs_5)))$.

$$cs' = LRU_{m_1}(cs_5) = \begin{cases} c_0 & [\{2,6\}, \{2,8\}, \{6,10\}, \{0\}] \\ c_1 & [\{3\}, \{7,9\}, \{11\}, \{1\}] \end{cases} \quad (11)$$

$$cs'' = LRU_{m_2}(cs') = \begin{cases} c_0 & [\{6,8\}, \{6,10\}, \{0\}, \{2\}] \\ c_1 & [\{3\}, \{7,9\}, \{11\}, \{1\}] \end{cases} \quad (12)$$

$$cs''' = LRU_{m_3}(cs'') = \begin{cases} c_0 & [\{6,8\}, \{6,10\}, \{0\}, \{2\}] \\ c_1 & [\{7,9\}, \{11\}, \{1\}, \{3\}] \end{cases} \quad (13)$$

The access of m_1 in equation 11 shows a reordering in cache set c_1 since $m_1 \in c_1[1]$. The access of $LRUm_2(cs')$ in equation 12 is more complicated to model because $m_2 \in c_1[1]$ as well as $m_2 \in c_1[2]$. Consider now all valid cache states that can be constructed from cs' for cache set c_0. The cache states $[2,2,6,0]$, $[2,2,10,0]$, $[6,2,6,0]$ and $[6,8,6,0]$ are not valid because one element occurs several times in the 4-way associative cache set. The original LRU replacement strategy results to:

$LRU_2[2,8,6,0] = [8,6,0,2] \quad LRU_2[2,8,10,0] = [8,10,0,2]$
$LRU_2[6,8,10,0] = [8,10,0,2] \quad LRU_2[6,2,10,0] = [6,10,0,2]$

The union of the above resulting cache states is then the conservative cache state cs'' that is computed by the algorithm in Figure 4

(lines 6-12). Finally $LRU_{m_3}(cs'')$ is applied in equation 13, which is a reordering for c_1. This example has shown the application of the analysis algorithm to a small control flow graph.

Note, that the set-based cache model may yield to an overestimation, because the model includes cache states that are invalid. However the representation is conservative, such that the actual cache related preemption delay is always smaller than the estimated one.

5. Response time analysis

This section gives an overview of our scheduling analysis, which was developed in previous work [18] [19]. We assume an activation model $E_j(R_i)$ that determines how often task τ_j is activated during a time interval R_i. The data flow analysis delivers the worst case cache related preemption delay for the nth most expensive preemption. We have proven the following Theorem in [19].

Theorem. Given a set of real-time tasks scheduled by a fixed priority preemptive policy where task τ_i suffers a worst case penalty $\delta^z_{j,i}$ by τ_j for the zth preemption, there either exists a worst case response time value for R_i for each task τ_i which makes the Equation system 14 to 16) true or such a task set is not schedulable.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} (E_j(R_i) \cdot C_j + \Delta_{ji}(R_i)) \qquad (14)$$

$\Delta_{ji}(R_i)$ is computed by the following formulas:

$$n = \sum_{k=j}^{i-1} E_k(R_i) \quad M = \biguplus_{k=i}^{j+1} \biguplus_{E_k(R_i)} \hat{\delta}_{j,k} \qquad (15)$$

$$\Delta_{ji} = \sum_{k=1}^{n} max^k(M) \qquad (16)$$

The term $E_k(R_i)$ denotes the maximum number of activations of τ_k during the interval R_i, $\hat{\delta}_{j,k} = \{\delta^1_{j,k}, \delta^2_{j,k}, \cdots, \delta^{E_j(R_k)}_{j,k}\}$ is the set of the $E_j(R_k)$ most expensive preemptions costs imposed by τ_j on τ_k, $max^k(M)$ returns the kth most largest value of the set M. The operator $\uplus$ symbolizes the union of two sets allowing multiple occurrence of elements, e.g. $\{a, b\} \uplus \{a, c\} = \{a, a, b, c\}$.

This scheduling analysis is polynomial, yet we have given some extensions to consider some cases of task phasing. The preemption delays are calculated interactively by three main modules. First the scheduling analysis assumes a default cache related preemption delay and performs a standard scheduling analysis. Then the maximum number of preemptions of task τ_i by τ_j are given by the number of activations and the period of the tasks. This is formulated as a scenario tuple $(\tau_{preempting}\tau_{preempted}, k)$.

At a second step the most expensive n preemption costs have to be computed. We use the control flow graph as the representation of task. An optimization algorithm that guarantees to find the optimal solution is necessary to find the k most expensive preemption points. The branch and bound algorithm is such an algorithm. It gets the scenario tuple as input and needs the actual costs of the kth preemption provided that $k-1$ preemptions have been occurred at $n_{i_1}, n_{i_2}, n_{k-1}$. A simple approximation would be to determine the k most expensive preemption points, but this is not accurate as shown in [18]. The branch and bound algorithm gets this data from the data flow analysis (DFA) which was described in the previous sections. The DFA gets as input a tuple of nodes where a preemption already took place and determined the preemption cost for the k preemption. Each individual preemption cost is computed by analyzing the maximum number of useful cache blocks of the preempted task and the maximum number of used cache blocks of the preempting task.

The data flow analysis in [18] was based on the cache state approach and therefore exponential in time complexity. This pitfall is overcome with this novel scalable precision analysis. Now we are able to integrate both, the task level analysis of cache behavior and the system level scheduling analysis in polynomial time.

6. Experiments

This section describes the experimental results for the scalable precision cache analysis. In the experiments the influence of cache size, utilization and associativity is evaluated for different embedded benchmarks which are mainly taken from [11] and [14]. Figure 1 presents their main memory usage in Byte [B], the number of C source code lines and the WCET in 10^3 clock cycles [clk] for a 4-way set associative 1KB instruction cache. The worst case execution time of each task was determined by a cycle accurate ARM945 processor simulator [2] for the different instruction cache architectures with a 30 cycles cache miss penalty. Each instruction is 4 byte long and we choose a cache blocks size of 8 byte, such that 2 instructions fit in a cache block. That these benchmarks are rather small compared to a real application curses us to use a smaller cache. But this is not a limitation, when the cache footprint is taken into consideration.

Id	Mem	C-Ln	WCET	Description
τ_1	376	83	1.401	square root calculation
τ_2	296	275	1.617	packet receiver
τ_3	888	180	15.34	fast fourier transform
τ_4	144	34	39.23	exchangesort
τ_5	1023	286	4051	whetstone

Table 1. Benchmark Description with Memory Usage[B], c-lines and WCET[$10^3 clk$].

The cache footprint index determines how many tasks use a single cache block on average. Table 2 shows this index for some task setups for a direct mapped cache and cache sizes of 256, 512, 1024 and 2048 Byte, respectively.

Tasks	256B	512B	1024B	2048B
τ_1 τ_2	2.0	1.68	1.34	0.75
τ_1 τ_3	2.0	1.68	1.31	0.83
τ_1 τ_4	1.53	0.95	0.47	0.24
τ_1 τ_5	2.0	1.68	1.34	1.17
τ_2 τ_3	2.0	2.0	1.97	1.23
τ_2 τ_5	2.0	2.0	2.0	1.57

Table 2. Cache footprint index for a direct mapped cache.

For example both tasks τ_1 and τ_2 occupy the entire 256B cache, hence the footprint index is 2. The footprint of 1.17 of τ_1, τ_5 for 2 KB cache describes the fact that most cache lines are used by one task only. In the case of a 2KB cache for tasks τ_1 and τ_4 we can study the cache effects for a large cache compared to small application size, since most of the cache lines are empty (index=0.24). The cache footprint index enables us to generalize the results of this paper, because the cache behavior mainly depends on its utilization, and not on cache size or application size alone.

We have implemented the analysis approach of the preceding sections for direct mapped and n-way associative instruction caches. We assume the distance metric of equation 8 and the strategy M2, which compares all possible cache cache states.

Figure 6 shows the preemption costs for several task scenarios for a 4-way set associative 512 Byte instruction cache. In the legend

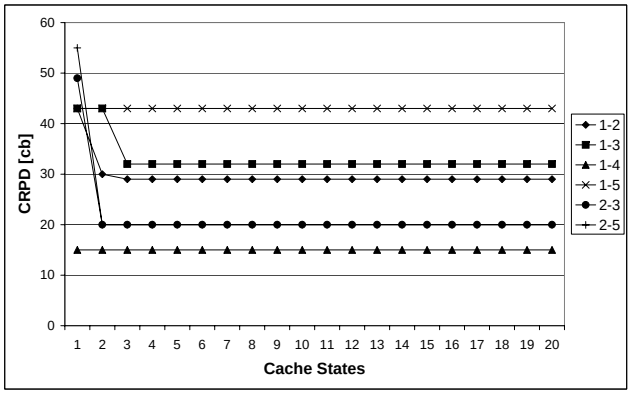

Figure 6. Preemption Delay for 4-way 512B cache.

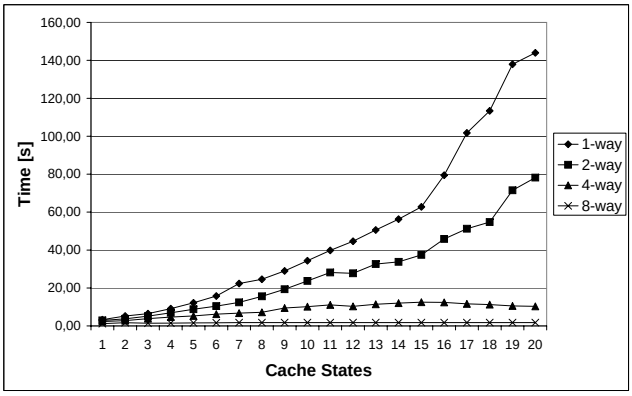

Figure 8. Analysis time (512B cache).

the preemption scenario is denoted as $1-2$ which means that task τ_1 preempts τ_2. The x-axis shows the number of cache states, Z, that are allowed during the data flow analysis and the y-axis shows the cache related preemption delay in cache blocks. For example, in the preemption scenario, where τ_2 preempts τ_5 the CRPD for $Z = 1$ is 57 and for $Z = 2$ is 20 cache blocks. The CRPD for the preemption scenario $2-3$ is 49 for $Z = 1$ and 20 cache blocks for $Z > 1$.

As the number of cache states increases, the CRPD drops by 72% for scenario $1-2$ and to 35% for scenario $2-5$. The greatest change is between $Z = 1$ and $Z = 2$, for greater Z the CRPD is constant. For some cases the CRPD does not decrease. The reason for the significant change for small numbers of Z is that during the analysis only a few number of RCS and LCS states were used.

Figure 7 presents the influence of associativity for the preemption scenario $2-3$ for a 512B cache. Again the cache states and the CRPD is shown on the x-axis and y-axis, respectively. The figure shows that for $Z \geq 3$ the CRPD is constant, except for the 2-way associative cache, where the jump happens between 7 and 8 cache states.

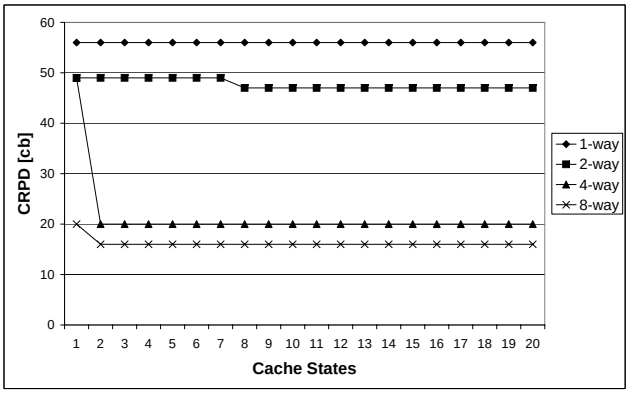

Figure 7. CRPD for several associativities (512B).

The analysis time for the same setup, 512B cache for scenario $2-3$, is presented in Figure 8. The curves show an exponential growth of analysis time for increasing number of cache states for direct mapped (1-way) and 2-way set associative caches. For higher order of associativities the analysis time is in a rage of 2 to 18 seconds. A direct mapped or 2-way associative cache consists of relatively high number of cache sets. Therefore the number of combinations is much greater then higher order of associativities.

From Figure 7 and Fig 8 we conclude that a higher number of cache states does not necessarily lead to reduced preemption delay. Already for small number of Z there is a significant jump of analysis precision. Analysis time potentially grows with number of cache states. A strategy is to analyze only for the first few number of states, until the preemption delay does not change, because further increase of Z is time consuming without an analysis improvement.

In Figure 9 we evaluate the effect of different 4-way associative cache sizes for the scenario $2-3$. The preemption delay increases for larger caches as more cache blocks can potentially be useful or can potentially be used by the preempting task. The difference between 1KB and 2KB cache size is small since the number of useful cache blocks is also bounded by the application.

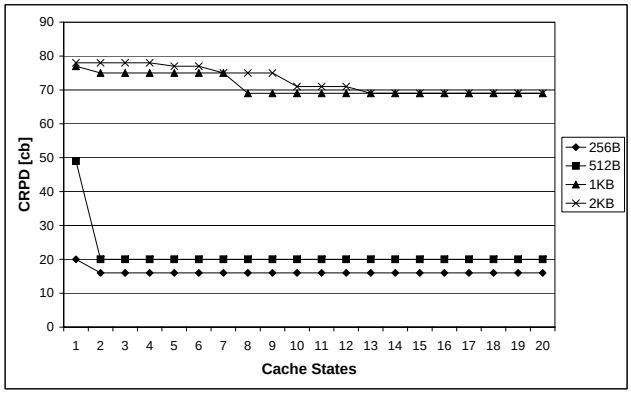

Figure 9. CRPD for different cache sizes (4-way).

Finally, we integrate the CRPD estimation of m-way associative instruction caches into a scheduling analysis. We compare our results to Busquets-Mataix [4] and Petters [16] approach. The approach by [12] with exponential number of in-equations would have been to time-consuming to re-implement for comparison purposes.

We compute the worst case response time for the task set $\tau_1, \tau_2, \tau_3, \tau_4$ with τ_1 as highest priority task and τ_4 as lowest priority task. The execution time of the whetstone benchmark was much greater than the other four, that why we left it out. Figure 10 shows the total preemption delay in clock cycles during the entire schedule for several cache sizes with each 4-way set associativity. Compared to Busquets the data flow analysis with scaling factor

15 shows an improvement of 57%, 35%, 22% and 31% for 256B, 512B, 1KB and 2KB cache respectively. Compared to Petters our analysis shows an improvement of 39%, 43%, 59% and 70% for the given cache sizes.

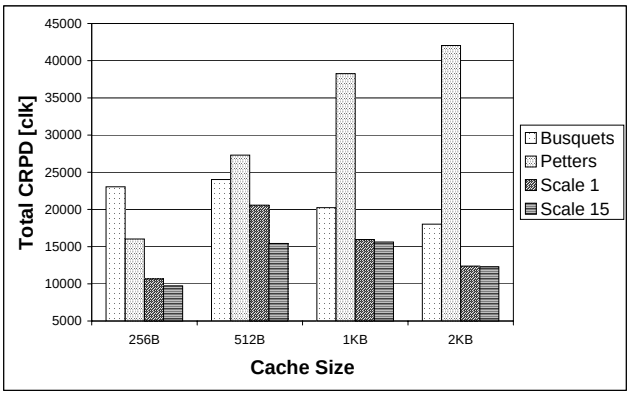

Figure 10. Total CRPD during response time of τ_4.

The response time of task τ_4 including cache related preemption delay is shown in Figure 11. The y-axis shows the response time in percentage of the response time that was calculated by Busquets approach. Besides Busquets, the results of Petter's approach, the set-based approach with scaling factor 1 and scaling factor 15 are draws. The analysis precision for scaling factor 15 improves between 5% for 1KB cache till 21% for 256 Byte cache.

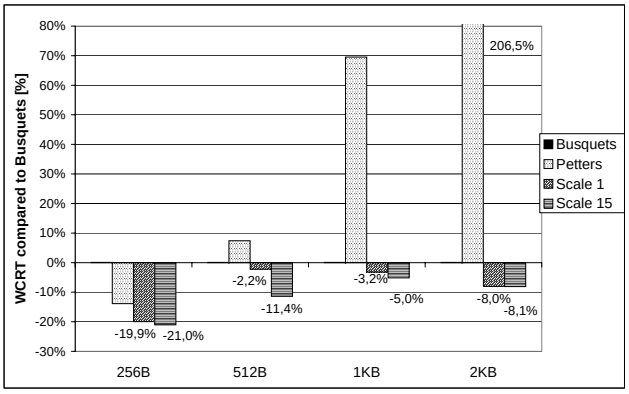

Figure 11. Comparison of WCRT for τ_4.

7. Conclusion

Cache memory introduces unpredictable interference to task execution time when it is used in real-time computing systems where preemptive scheduling is allowed. We have proposed a scheduling analysis that takes such interference into account. In the first step, the cache related preemption delay was computed by a scalable analysis framework for direct mapped caches. This offers the designer a valuable tool to find the tradeoff between analysis precision and analysis time.

In a second step we have extended this modeling conservatively to set-associative caches, while current approaches assume direct mapped caches or propose non conservative approximations. The results show that for small number of cache states the analysis finishes within seconds making it attractive for fast design space explorations. For later re-evaluation this parameter can be adjusted to obtain more accurate results which might take considerably more time. Finally we integrated the preemption delay analysis in a polynomial scheduling analysis which is as precise as other polynomial algorithms and which includes some aspects of exponentially complex analysis approaches.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, GB, 1988.

[2] ARM Developer Suite, (ADS) version 1.2. *http://www.arm.com*.

[3] A. Burns. *Preemptive priority based scheduling*, chapter An appropriate engineering approach, pages 225–248. Prentice-Hall International, Inc., 1994.

[4] J. V. Busquets-Mataix and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 204–212, June 1996.

[5] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE Real-Time Embedded System Workshop*, December 2001.

[6] A. Datta, S. Choudhury, A. Basu, H. Tomiyama, and N. Dutt. Satisfying timing constraints of preemptive real-time tasks through task layout technique. In *IEEE VLSI Design*, pages 97–102, January 2001.

[7] S.-S. L. et al. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–603, July 1995.

[8] Infineon. Tricore 1 manual http://www.infineon.com.

[9] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal (British Computer Society)*, 29:390–395, Oct. 1986.

[10] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium*, pages 229–239, 1989.

[11] C.-G. Lee, J. Hahn, and Y.-M. S. et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on computers*, 47(6):700–713, June 1998.

[12] C.-G. Lee, K. Lee, and J. H. et al. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on software engineering*, 27(9):805–826, November 2001.

[13] J. Liedtke, H. Härtig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *RTAS*, Montreal, Canada, June 9-11 1997.

[14] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS'03*, Newport Beach, CA, USA, Oct. 2003.

[15] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers,Norwell, MA, 1999.

[16] S. M. Petters and G. Färber. Scheduling analysis with respect to hardware related preemption delay. In *Workshop on Real-Time Embedded Systems*, London, UK, Dec. 2001.

[17] I. Puaut and D. Decotigny. Low-complexity algorihtms for static cache locking in multitasking hard real-time systems. In *RTSS*, 2002.

[18] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT*, Pisa, Italy, Sept. 2004.

[19] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *submitted to EUROMICRO Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[20] Symta/S - symbolic timing analyis of systems. www.symta.org.

[21] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*, 2000.