Scalable Precision Cache Analysis for Real-Time Software

JAN STASCHULAT and ROLF ERNST
Technical University of Braunschweig

Caches are needed to increase the processor performance, but the temporal behavior is difficult to predict, especially in embedded systems with preemptive scheduling. Current approaches use simplified assumptions or propose complex analysis algorithms to bound the cache-related preemption delay. In this paper, a scalable preemption delay analysis for associative instruction caches to control the analysis precision and the time-complexity is proposed. An accurate preemption delay calculation is integrated into a cache-aware schedulability analysis. The framework is evaluated in several experiments.

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

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Worst-case execution time analysis, cache, embedded systems, preemptive scheduling

ACM Reference Format:

Staschulat, J. and Ernst, R. 2007. Scalable precision cache analysis for real-time software. ACM Trans. Embedd. Comput. Syst. 6, 4, Article 25 (September 2007), 39 pages. DOI = 10.1145/1274858.1274863 http://doi.acm.org/10.1145/1274858.1274863

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 already difficult to determine cache behavior for a single task, it becomes even more complicated when 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

Authors' addresses: Technical University of Braunschweig, Hans Sommer Str. 66, D-38106 Braunschweig, Germany; e-mail: staschulat|ernst@ida.ing.tu-bs.de

individual tasks. This has been investigated in Kirk [1989], Mueller [1995], Liedtke et al. [1997], and Busquets et al. [1997]. The advantage is that cache blocks 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, as suggested in Datta et al. [2001] for instruction caches, aim at minimizing the intertask interference. Another approach is to lock frequently used cache blocks. Such techniques have been investigated by Kirk [1989], Campoy et al. [2001], Puaut and Decotigny [2002], and Campoy et al. [2005]. Both approaches increase area and power cost as they require larger caches or background memories to become effective. Therefore, heterogeneous memory architectures with caches and scratch-pad SRAM have been introduced in the TriCore architecture [Infineon 2004], where a scratch-pad can hold frequently used cache blocks. Compiler techniques for such architectures have been proposed by Panda et al. [1999]. 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 timing-analysis problem, where all tasks share a single cache, which is critical for larger systems of tasks.

On solution to this problem is to extend the general schedulability analysis by an additional term for cache-related preemption delays. The problem is twofold: first, the calculation of the time delay for reloading replaced cache blocks because of a single preemption; and second, a cache-aware schedulability analysis, which estimates the number of occurences of preemptions. In Busquets-Mataix and Wellings [1996], Lee et al. [1998], and Petters and Färber [2001] cache-aware response time analyses are described, which use a pessimistic estimation for a single preemption. Either the preempted task is considered in Lee et al. [1998], and Petters and Färber [2001] or the preempting task Busquets-Mataix and Wellings [1996] is considered. Recently, two approaches Lee et al. [2001] and Mitra et al. [2003] have been proposed in which the preempting as well as the preempted task is considered. Both approaches rely on data flow techniques, but use different cache representations.

The approach by Mitra et al. [2003] requires a greater time complexity, but is more precise than the Lee et al. [2001] approach. During system design, engineers must focus on different abstraction levels of system representation and have to cope with different precisions of timing behavior. In design space exploration, rough estimates of the system performance are sufficient, while accurate estimates are necessary in the very last steps, of real-time verification.

The system design process already involves many steps and real-time performance verification is one aspect. The design process would be impeded if too many different techniques were employed. Unprecise cache behavior estimations lead to an overdimensioned and inefficient design, while long verification times limit the design, space exploration.

1.1 Contributions

This paper addresses the problem of cache interference in a real-time embedded system, where all tasks share the same instruction cache. Preempitve

scheduling leads to frequent task interferences, which requires reloading of cache blocks. The key contributions of the proposed analysis framework are:

- A scalable precision cache analysis for preemption delay calculation. It combines the strengths of both approaches of Mitra et al. [2003] and Lee et al. [2001] by controlling the problem complexity and, therefore, the analysis precision. Thus the same analysis methodology can be used in early design phase, as well as in late time-intensive verification phase during system development.
- The analysis supports direct-mapped, as well as associative instruction caches.
- The analysis of a single task preemption is highly accurate by considering the preeempting as well as the preempted task.
- The scalable preemption delay calculation is integrated in a cache-aware response time analysis, that has been published in Staschulat et al. [2005] to improve the accuracy of total response times.
- Finally, the applicabiliy of the method is demonstrated in several experiments by comparing the analyis precision to previously published work and by evaluating the time and memory consumption.

1.2 Limitations

Data caches are not considered. A timing analysis for data caches has been proposed, e.g., in Vera, et al. [2003], Ramaprasad and Mueller [2005], and Staschulat and Ernst [2006] and a preemption delay analysis for data caches in Ramaprasad and Mueller [2006].

The approach is based on the control-flow graph of a task, which is constructed from the source code. Often software is secured by IP rights and is not available in source code. In this case, a control-flow graph of the application with associated instruction addresses could be an interface to our cache-analysis technique.

As the control-flow graph is used to represent the task execution, we only consider in-order issuing single processors. Timing analysis of out-of-order pipelines and parallel resource allocation can lead to timing anomalies [Schneider 2000] and are more difficult to analyze.

As in all other previous work, we assume a constant cache miss penalty, which is determined by the worst case access time to background memory. However, the average memory access time might be much smaller, depending on bus load and memory controller.

1.3 Overview

This paper is structured as follows. Related work is reviewed in Section 2. A motivating example is given in Section 3. We then present the new cache model in Section 4. We present the scalable precision cache analysis first for direct-mapped instruction caches in Section 5 and then for associative-instruction caches in Section 6. In Section 7, the overall framework for real-time analysis

is described. Experiments are presented in Section 8 before we conclude in Section 9.

2. RELATED WORK

2.1 Preemptive Scheduling and Schedulability Analysis

When multiple tasks share one resource, then two or more tasks may request the resource at the same time. Scheduling resolves these conflicting requests. Three major classes for task-scheduling strategies can be distinguished: First, nonpreemptive static execution is mainly used in highly regular digital signal processing. Second, priority-driven scheduling is often used in highly reactive systems operating in dynamic environments. The scheduling strategy can use fixed priorities or dynamic priorities. In the fixed-priority case, priority assignments often follow a rate-monotonic scheduling (RMS) [Liu and Layland 1973] or deadline-monotonic scheduling [Audsley et al. 1991] strategy; in the dynamic-priority case, priority assignments often follow an earliest-deadline-first (EDF) strategy. Finally a third class are preemptive time-slicing techniques, which are used for a fair distribution of a resource among tasks. In this paper, we investigate cache behavior effects for fixed-priority rate-monotonic scheduling.

Embedded applications often require a real-time behavior, which denotes the minimum and maximum time delays that are permitted for correct functionality. The importance of finishing a task within the predefined time window distinguishes soft and hard real-time system. While soft real-time systems accept frequent violations, hard real-time systems require that every software task must finish before a predefined time deadline. Schedulability analysis is a technique to verify that each task finishes before its deadline.

Several schedulability-analysis techniques have been proposed for fixed-priority preemptive scheduling [Tindell et al. 1994; Lehoczky et al. 1989; Joseph and Pandya 1986; Liu and Layland 1973]. Liu and Layland [1973] show that the rate-monotonic priority assignment, where a task with a shorter period is given a higher priority, is optimal when task deadlines are equal to their periods. They also give the following sufficient condition for schedulability for a task set consisting of n periodic tasks $\tau_1, \cdots, \tau_n$:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right), \tag{1}$$

where C_i is the worst-case execution time (WCET) of τ_i and T_i its period.This condition states that if the total utilization of the task set U is lower than the given utilization bound ($n(2^{1/n}-1)$), the task set is guaranteed to be schedulable under the rate-monotonic priority assignment. Later Lehoczky et al. [1989] develop a necessary and sufficient condition for schedulability based on utilization bounds.

Because of the maximum utilization bound of 0.69 of the above approach, Pandya et al. [Joseph and Pandya 1986] and Tindell et al. [1994], have developed an iterative response-time analysis. 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 allocates in a time window R_i the task τ_i's worst-case execution 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 R_i^n, resulting in a longer time window R_i^{n+1} that might include greater interference in the next step. The process is finished when the window stops growing ($R_i^{n+1} = R_i^n$). If the resulted response time for any task is greater than its deadline ($R_i^{n+1} = R_i^n = R_i > D_i$), the task-set is not schedulable. The iterative relation is shown in Eq. (2) in which $hp(i)$ denotes the set of tasks with a higher priority than task τ_i.

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

2.2 Cache-Aware Schedulability Analysis

At a context switch, cache blocks can be replaced by higher priority tasks and require a cache reload increasing the response time. Measurement-based approaches to estimate the response time, which consider cache-related preemption delays, have been proposed by Mogul and Borg [1991], Corti et al. [2000], and [Sebek 2001], but safe and accurate bounds can not be guaranteed, in general, because simulation results strongly depend on the code coverage level reached by input data.

As an alternative, cache-related time delays resulting from a preemption have been integrated into schedulability analysis. Basumallick and Nilsen [1994] extend Liu and Layland [1973] schedulability condition by an additional term for the cache interference. The actual time delay for cache interference is not further analyzed. One drawback of such a technique is that it suffers from the pessimistic utilization bound 0.69 for larger task sets. Many task sets that have a total utilization higher than this bound can be successfully scheduled [Lehoczky et al. 1989].

Therefore, the approach in Busquets-Mataix and Wellings [1996] extends the response-time approach of Tindell et al. [1994] by an additional term γ_j that denotes the cache-related preemption delay:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot (C_j + \gamma_j) \tag{3}$$

The term γ_j considers only the used cache blocks of the preempting task. However, it is possible that a replaced cache block by the preempting task is one that is no longer needed by the preempted task or one that will be replaced without being referenced again during a usual execution of the preempted task. These cache blocks are known as *useful* cache blocks [Lee et al. 1998]. In Petters et al. [Petters and Färber 2001] a response-time analysis is presented that considers only the useful cache blocks of a preempted task. The quantity of useful cache

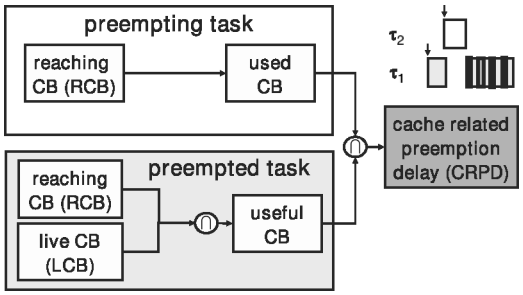

Fig. 1. Methodology to compute the cache related preemption delay for two tasks.

blocks is approximated by a constant percentage of all used cache blocks. In Lee et al. [1998] a data flow technique is presented that calculated the set of useful cache blocks and a cache-aware schedulability analysis is presented.

2.3 Preemption Delay Calculation with Data Flow Analysis

As a major step forward, both tasks have been considered. The set of useful cache blocks of the preempted task and the set of used cache blocks of the preempting task have been analyzed by Lee et al. [1998, 2001] and Mitra et al. [2003].

The problem of preemption delay calculation is split into three parts. First, the number of useful cache blocks of the preempted task τ is computed by two iterative data flow analysis (reaching cache blocks (RCS) and live cache blocks (LCS)). Second, the maximum number of used cache blocks of the preempting task τ' is computed by an iterative data flow analysis (reaching cache states at last node of control-flow graph RCS_{end}). Finally, the maximum intersection of useful cache blocks of the preempted task τ and used cache blocks of preempting task τ' bounds the cache-related preemption delay. This methodology is shown in Figure 1.

The reaching cache state $RCS[B]$ at a basic block B contains all possible cache blocks when B is reached via any incoming program path. The live cache state *LCS[B]* at a basic block B contains all the first memory references to cache blocks via any outgoing program path from B. *RCS[B]* capture (potentially) available cache blocks when the task is preempted and *LCS[B]* capture (potentially) re-accessed cache cache blocks when the task resumes execution. In both approaches iterative data flow techniques are used to calculate the RCS and LCS properties. We briefly review the general concept of data flow technique for the computation of RCS. The calculation of LCB is analogous.

To compute RCS_B, the quantities $RCS_{in}[B]$ and $RCS_{out}[B]$ are computed as a least fixed point. Once the fixed point is reached, $RCS[B] = RCS_{out}[B]$. Initially we set $RCS_{in}[B] = \emptyset$ and $RCS_{out}[B] = gen[B]$. For each basic block B, the *gen*-set contains the *last* memory blocks that are loaded to the cache during the execution of basic block B. The definition of the *gen* set depends on the underlying cache model and is, therefore, different in Lee and Mitra approach.

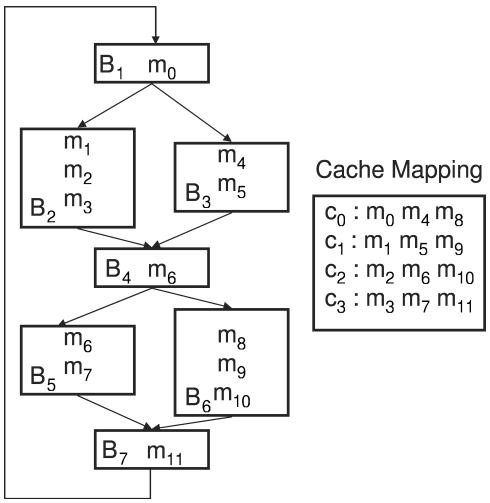

Fig. 2. Control flow graph with memory blocks and cache mapping for direct-mapped instruction cache with four cache blocks.

The iterative equations are as follows:

$$RCS_{in}[B] = \bigcup_{p \in pred(B)} RCS_{out}[p] \tag{4}$$

$$RCS_{out}[B] = \{r \odot gen[B] | r \in RCS_{in}[B]\} \tag{5}$$

$$c \odot c' = \begin{cases} c' & if \quad c' \neq \perp \\ c & otherwise \end{cases} \tag{6}$$

The $\odot$ operation is defined in Eq. (6) over memory blocks m and m'. It is extended to cache states by applying the operation to each cache set individually. A data flow algorithm, as described in Aho et al. [1988] and Lee et al. [1998], are used to calculate this fixed point.

While in Lee's approach the cache contents is represented by a set of memory blocks; in Mitra' approach the cache contents is represented by a cache state of memory blocks. The different representation causes a low-time complexity time complexity set-based in Lee's approach versus a higher time complexity in Mitra's approach In Section 3, we compare both approaches and motivate the scalable precision cache analysis.

3. MOTIVATIONAL EXAMPLE

We motivate our scalable precision analysis with a comparison of Lee's and Mitra's approach for the *RCS* calculation.

A task is represented by its control-flow graph (CFG) where nodes represent basic blocks and edges specify the control flow between basic blocks. An example control flow graph is shown in Figure 2. It shows a a loop statement with two `if-then-else` statements. A node B_i lists the memory blocks that correspond to the assembly instructions of basic block B_i. For example, the memory blocks m_1, m_2, and m_3 are loaded to the cache during execution of basic block B_2. For this

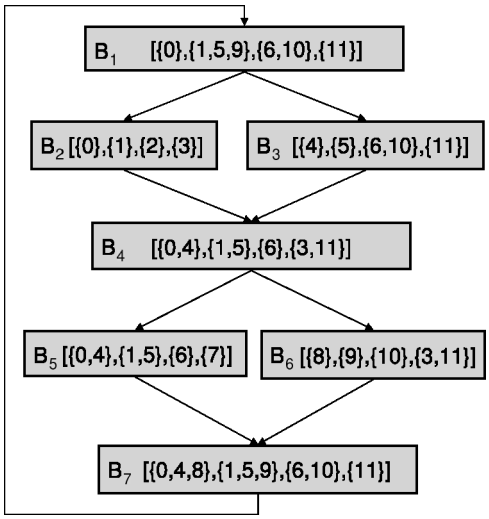

Fig. 3. RCS calculation by Lee.

Table I. Gen-Sets for the Flow Graph in Figure 2

B_i	$gen[B_i]$	B_i	$gen[B_i]$
B_1	[{0}, {}, {}, {}]	B_5	[{}, {}, {6}, {7}]
B_2	[{}, {1}, {2}, {3}]	B_6	[{8}, {9}, {10}, {}]
B_3	[{4}, {5}, {}, {}]	B_7	[{}, {}, {}, {11}]
B_4	[{}, {}, {6}, {}]		

example, we assume a direct-mapped cache with four cache sets. The mapping of memory blocks to cache sets is also given in Figure 2.

3.1 Set-Based Approach by Lee

The approach by Lee uses a *set of memory blocks* to store multiple memory blocks in a cache set. Figure 3 shows the *RCB* sets after the data flow analysis has converged. Table I summarizes the *gen* sets for the example. To abbreviate the notation we use only the index of memory blocks. The notation of the cache state at B_4 [{0, 4}, {1, 5}{6}, {3, 11}] represents a cache in which m_0 and m_4 are available in cache set c_0, memory blocks m_1 and m_5 are available in cache set c_1, m_6 is available in cache set c_2, and m_3 and m_{11} are available in c_3. In the data flow analysis, the contents of each cache set is propagated via the edges and is merged for each cache set.

The data flow algorithm is demonstrated on the example flow graph of Figure 2. Initially all $RCS^c_{in}[B]$ are empty and $RCS^c_{out}[B]$ are initialized with the $gen^c[B]$ sets. The results are summarized in Table II for the first and second iteration. The RMB sets of the third iteration are the same as in the second iteration; therefore, they are omitted.

We explain the calculation of *RCS* of Eq. (4–6) for the second iteration at basic block B_4. In this case, the RCS_{out} of B_2 and B_3 are merged and cache set

Table II. Reaching Cache Blocks $RCS_{out}[B_i]$ for Set-Based Approach by Lee

B_i	$RCS_{out}(B_i)$ 1st Iteration	$RCS_{out}(B_i)$ 2nd Iteration
B_1	[{0}, ∅, ∅, ∅]	[{0}, {1, 5, 9}, {6, 10}, {11}]
B_2	[{0}, {1}, {2}, {3}]	[{0}, {1}, {2}, {3}]
B_3	[{4}, {5}, ∅, ∅]	[{4}, {5}, {6, 10}, {11}]
B_4	[{0, 4}, {1, 5}, {6}, {3}]	[{**0**, **4**}, {**1**, **5**}, {**6**}, {**3**, **11**}]
B_5	[{0, 4}, {1, 5}, {6}, {7}]	[{0, 4}, {1, 5}, {6}, {7}]
B_6	[{8}, {9}, {10}, {3}]	[{8}, {9}, {10}, {3, 11}]
B_7	[{0, 4, 8}, {1, 5, 9}, {6, 10}, {11}]	[{0, 4, 8}, {1, 5, 9}, {6, 10}, {11}]

Table III. Reaching Cache States $RCS_{out}[B_i]$ for State-Based Approach by Mitra

B_i	$RCS_{out}[B_i]$ 1st Iteration	$RCS_{out}[B_i]$ 2nd Iteration
B_1	[0, ⊥, ⊥, ⊥]	[0, 1, 6, 11], [0, 9, 10, 11], [0, 5, 6, 11]
B_2	[0, 1, 2, 3]	[0, 1, 2, 3]
B_3	[4, 5, ⊥, ⊥]	[4, 5, 6, 11], [4, 5, 10, 11]
B_4	[0, 1, 6, 3], [4, 5, 6, ⊥]	[**0**, **1**, **6**, **3**], [**4**, **5**, **6**, **11**]
B_5	[0, 1, 6, 7], [4, 5, 6, 7]	[0, 1, 6, 7], [4, 5, 6, 7]
B_6	[8, 9, 10, 3]	[8, 9, 10, 3], [8, 9, 10, 11]
B_7	[0, 1, 6, 11], [4, 5, 6, 11], [8, 9, 10, 11]	[0, 1, 6, 11], [4, 5, 6, 11], [8, 9, 10, 11]

c_2 is replaced with $gen^{c_2}[B_4]$:

$$\begin{aligned} RCS_{in}[B_4] &= [\{0\}, \{1\}, \{2\}, \{3\}] \cup [\{4\}, \{5\}, \{6, 10\}, \{11\}] \\ &= [\{0, 4\}, \{1, 5\}, \{2, 6, 10\}, \{3, 11\}] \\ RCS_{out}[B_4] &= [\{0, 4\}, \{1, 5\}, \{\mathbf{6}\}, \{3, 11\}] \end{aligned}$$

3.2 State-Based Approach by Mitra

As an alternative, the approach by Mitra [Mitra et al. 2003] uses several *cache states* when more memory blocks are available in a cache set. The output for the *RCS* calculation is shown in Table III and graphically represented for the second iteration in Figure 4.

A cache state consists of cache sets, that are either empty (⊥) or contain a single memory block m_i. The data flow analysis is, again, described at basic block B_4: The contents for all cache sets is given in the following:

$$\begin{aligned} RCS_{in}[B_4] &= \{[0, 1, 2, 3], [4, 5, 6, 11], [4, 5, 10, 11]\} \\ RCS_{out}[B_4] &= \{[0, 1, \mathbf{6}, 3], [4, 5, \mathbf{6}, 11]\} \end{aligned}$$

Memory block m_6 is mapped to cache set c_2. Note, that the number of cache states reduces because the duplicated cache states are removed ([4, 5, 6, 11]). Since $gen[B_4] = [\{\}, \{\}, \{6\}, \{\}]$, the third cache set, containing $m_2, m_6, and\, m_{10}$, is replaced by m_6. In this approach, the cache states are duplicated if some cache sets are not equal, resulting in an increased number of cache states. For example, node B_5 and B_6 contain two cache states, but basic block B_7 has three cache states. This increase of cache states scales exponentially with the number of branches in a task. This higher time-complexity comes with the gain of a higher analysis precision.

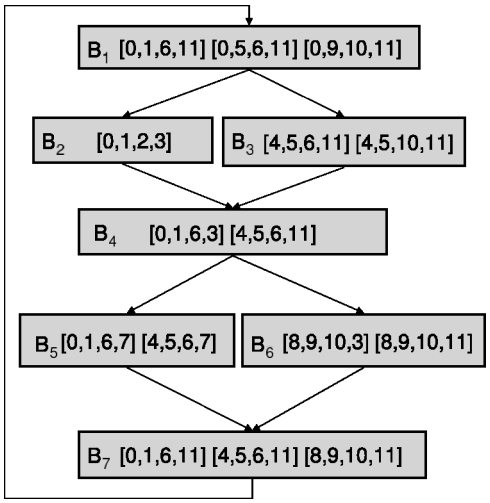

Fig. 4. RCS calculation by Mitra.

3.3 Comparison and Discussion

To directly compare the approaches, we calculate the useful cache blocks $USE[B_4]$ for basic block B_4. These results were taken from Tables II and III. The computation of LCS has not been shown because of space requirements, but the computation is analogous to the RCS calculation. The set of useful cache blocks calculated by Lee's approach $USE_{lee}[B_4]$ is given by:

$$\begin{aligned} RCB_{out}[B_4] &= [\{0,4\},\{1,5\}\{6\},\{3,11\}] \\ LCB_{out}[B_4] &= [\{0,8\},\{1,5,9\}\{6,10\},\{7,11\}] \\ USE_{lee}[B_4] &= RCB_{out}[B_4] \cap LCB_{out}[B_4] = [\{0\},\{1,5\}\{6\},\{11\}] \end{aligned} \tag{7}$$

Assuming that all useful cache blocks are used by the preempting task, the total preemption delay would be four cache blocks. The set of useful cache blocks calculated by Mitra's approach $USE_{mitra}[B_4]$ is given by:

$$\begin{aligned} RCS_{out}[B_4] &= \{[0,1,6,3],[4,5,6,11]\} \\ LCS_{out}[B_4] &= \{[8,9,10,11][0,1,6,7][0,5,6,7]\} \\ USE_{lee}[B_4] &= \max(RCB_{out}[B_4] \cap LCB_{out}[B_4]) \\ &= [0,1,6,3] \cap [0,1,6,7] = [0,1,6,\perp] \end{aligned} \tag{8}$$

Assuming that all useful cache blocks are removed by the preempting task, the total cache-related preemption delay would be three cache blocks. This is one cache block less, or 25% in relative terms, than in Lee's approach. The number of useful cache blocks are shown in Table IV for all basic blocks. In most cases the number of useful cache blocks computed by state-based approach is smaller than the set-based approach. The reason for a higher precision is the greater number of cache states that captures execution path information.

To conclude, we have demonstrated why the state-based approach by Mitra. is more precise, but has a higher time complexity than the set-based approach

Table IV. Useful Cache Blocks of State-Based Approach by Mitra USE_{mitra} and of Set-Based Approach by Lee USE_{lee}.

Basic block	USE_{mitra}	USE_{lee}
B_1	**3**	4
B_2	2	2
B_3	**2**	3
B_4	**3**	4
B_5	**2**	3
B_6	1	1
B_7	**3**	4

by Lee. The question arises, whether the analysis precision of the state-based approach could be accomplished with a fewer number of states? If not, the same analysis precision can be reached. How then does the precision scale with the time complexity? How many cache states would be necessary for sufficiently accurate results?

To answer these questions, we propose an scalable precision cache analysis that limits the number of cache states at each node. Whenever the number of cache states is larger then a given bound, cache states are merged. This technique bounds, therefore, the time complexity, because the number of cache states is bounded but possibly reduced the analysis precision. A new cache model, which is necessary to provide a sufficiently general data structure, is described in the next section.

4. SCALABLE PRECISION CACHE MODEL

In the following, we define a cache state c and a cache set c_i that are used in the scalable precision cache analysis for direct-mapped and associative-instruction caches.

Definition. A cache state c is defined as a vector of cache sets c_i: $c = [c_1, \cdots, c_S]$ where S denotes the total number of cache sets.

Definition. A cache set c_i is defined as a vector of sets of memory blocks M: $c_i = [M, \cdots, M]$. The length of the vector is given by the associativity n of the cache. The cache set $c_i[1]$ for a direct-mapped cache at execution point p contains the memory blocks $m_1, \cdots, m_k$ if these memory blocks may have been mapped to c_i at execution point p: $c_i[1] = \{m_1, \cdots m_k\}$, otherwise $c_i = \emptyset$. Analogously, the nth element of cache set $c_i[n]$ for associative caches is defined as containing the nth most recently used cache blocks.

For a direct-mapped cache, the vector c_i has exactly one element, because there is only one cache set to which a memory block can be mapped to: $c_i = [M]$. For example, a cache state of a direct-mapped cache with four sets is defined as:

$$c = [[M], [M], [M], [M]] \tag{9}$$

As a second example, a cache state for a two-way associative cache with four sets is defined as

$$c = [[M, M], [M, M], [M, M], [M, M]] \tag{10}$$

The usage of all memory blocks M for each cache set can be refined for complexity considerations, because not every memory block can be mapped to every cache set. Cache set c_i contains only those memory blocks that map to this cache set, e.g., $c_i = [M_i, M_i, M_i, M_i]$ in which M_i denotes the set of memory blocks that map to cache set c_i. This definition allows us to tighten time and space complexity considerations, because the number of elements of M_i is given by $|M_i| = \frac{|M|}{S}$.

5. PREEMPTION DELAY ANALYSIS FOR DIRECT-MAPPED CACHES

Data flow techniques to bound the total preemption delay have been proposed by the state-based approach by Mitra et al. [2003] and the set-based approach by Lee et al. [1998]. While the approach by Lee simplifies the notion of a cache contents by using sets, which has the advantage of a low time complexity, the approach by Mitra distinguishes between different execution paths by using cache states, which potentially leads to a large number of cache states at each basic block. Our approach combines the strengths of both approaches by limiting the number of cache states at each basic block, thereby scaling the time complexity as well as the analysis precision. Whenever the threshold of the maximum number of cache states is exceeded, cache states are merged, reducing the time complexity, but also reducing the accuracy of cache content prediction. We start by describing the scalable data flow analysis.

5.1 Scalable Data Flow Analysis

Based on the general data flow analysis, as described in Section 2.3, we present the new scalable precision cache analysis.

5.1.1 *Computing Reaching Cache States.* We adopt the presentation of the iterative data flow algorithms from Mitra. For the RCS property, we define $RCS_{in}[B]$ and $RCS_{out}[B]$ as the cache state of reaching cache blocks just before and just after the execution of basic block B. After the fixed point is reached, we set $RCS[B] = RCS_{out}[B]$. Initially

$$RCS_{in}[B] = \emptyset \quad RCS_{out}[B] = gen[B] \tag{11}$$

For each basic block B, the $gen[B]$ is defined as a vector $gen[B] = [[M_0], \cdots, [M_{n-1}]]$, where $M_i = \{m\}$ if m is the *last* memory block in B that maps to cache block i and $\emptyset$ if no memory block in B maps to cache block i. Thus, $gen[B]$ represents all the memory blocks that are available in the cache at the end of the execution of basic block B. The iterative Eqs. 4–6 are modified as follows:

$$RCS_{in}[B] = bound_Z\left(\bigcup_{p \in pred(B)} RCS_{out}[p]\right) \tag{12}$$

$$RCS_{out}[B] = \{r \odot gen[B] | r \in RCS_{in}[B]\} \tag{13}$$

$$c \odot c' = \begin{cases} c' & if \quad c' \neq \emptyset \\ c & otherwise \end{cases} \tag{14}$$

```
// input: set of cache states C
// output: set of cache states C with |C| ≤ Z
void bound(Z, C) {
1       while (|C| > Z) do {
2           choose (c_i, c_j) with d(c_i, c_j) minimal, c_k ∈ C
3           C = C \ {c_i} \ {c_j}
4           C = C ∪ {c_i ⊙ c_j}
5       }
6 }
```

Fig. 5. $bound_Z(C)$ algorithm.

For the proposed cache model, $c \odot c'$ denotes a binary operation on memory blocks M and is applied for each cache set. The cache set cs_i is represented by a vector with a single element: $cs_i = [M]$ thus $c = cs_i[1]$ and $c' = cs'_i[1]$. If the $gen[B]$ set is not empty, then the result is $gen[B]$, otherwise the set of memory blocks is replaced by $RCS_{in}[B]$. The function $bound_Z(C)$ reduces the number of total cache states of C to Z elements, where C is a set of cache states. Its implementation is described in Section 5.2

5.1.2 *Computing Live Cache States.* Similarly the LCS property is computed by an iterative fixed-point algorithm. The only difference is that the $LCS_{out}[B]$ is defined in terms of $LCS_{in}[B]$ of all successors of basic block B: Initially,

$$LCS_{out}[B] = \emptyset \quad LCS_{in}[B] = gen[B] \tag{15}$$

For each basic block B, $gen[B]$ is defined as $gen[B] = [[M_0], \cdots, [M_{n-1}]]$, where $M_i = \{m\}$ if m is the *first* memory block in B that maps to cache block i and $\emptyset$ if no memory block in B maps to cache block i. The iterative equations are:

$$LCS_{out}[B] \;=\; bound_Z\left(\bigcup_{s \in succ(B)} LCS_{in}[s]\right) \tag{16}$$

$$LCS_{in}[B] \;=\; \{l \odot gen[B] | l \in LCS_{out}[B]\} \tag{17}$$

The operation $\odot$ is defined as in the computation of RCS_B.

5.2 Bounding Number of Cache States

The number of cache states is bounded with a function $bound_Z$. The function $bound_Z(C)$ reduces the number of states of set C to Z elements, if $|C| > Z$, otherwise $bound_Z(C) = C$. The idea is to merge those cache states, which are almost equal. We formalize this idea by using a distance metric over cache states that characterizes the number of different elements of two cache states. We then repeatedly choose two elements with minimum distance and merge them, until the total number of elements in C is equal to Z. The objective is to reduce the number of cache states while keeping them as different as possible.

The $bound_Z$ algorithm is shown in Figure 5. In line 2, two elements $c_i, c_j \in C$ with the minimum distance $min\{d(c_k, c_l) | c_k, c_l \in C\}$, are chosen. In line 3, these elements are removed from C. In line 4, the merged cache state $c_i \cup c_j$ is inserted

to C. Therefore, the number of elements of C decreases by one in each iteration and, thus, the algorithm always terminates.

5.2.1 *Distance Metric.* The function $d(a, b)$ of two cache states a, b is defined as a metric that delivers the difference of two cache states. Several metrics are possible. We present a simple metric d_1 and a more complex metric d_2. The simple metric d_1 only counts the number of different cache sets, ignoring how many elements in each set are different. This is shown in Eq. (18) where S denotes the total number of cache sets.

$$d_1(a, b) = \sum_{k=1}^{S} \begin{cases} 1 & \textit{if} \quad a_k[1] \neq b_k[1] \\ 0 & \textit{otherwise} \end{cases} \tag{18}$$

The variable $a_k[1]$ denotes the set of memory addresses of cache set c_k. The scalable precision cache model for a direct-mapped cache represents each cache set as a vector with one set (refer to Section 4). thus, we have to use the first element of the vector. The complexity of d_1 is

$$O(d_1) = O(S \cdot X) \tag{19}$$

where X denotes the maximum number of elements of the set $a_k[1]$. The equality test of two sets can be computed in linear time. We use the O notation to express the time complexity [Cormen et al. 1997]. A bound for the maximum number of memory blocks X is discussed in Section 5.3.3.

As an alternative, a more sophisticated metric d_2 is proposed to count the number of different memory blocks of each cache set with the symmetric difference. The definition of d_2 is given in Eq. (20).

$$d_2(a, b) = \sum_{k=1}^{S} \left| (a_k[1] \cup b_k[1]) \setminus (a_k[1] \cap b_k[1]) \right| \tag{20}$$

The complexity of d_2 is given by taking S times the union of two cache sets with X elements; by taking the intersection S times of two sets with, at most, X elements each and by taking the set difference of a set with $2X$ elements and X elements

$$O(d_2) = O(S \cdot (t_{\cup}(X, X) + t_{\cap}(X, X) + t_{\mathit{setdiff}}(2X, X)) = O(S \cdot X^2) \tag{21}$$

5.2.2 *Cache State Selection.* There are several selection strategies to choose candidates of cache states of C. One strategy, Sel_1, is to maintain only one merged set, such that the metric is computed between pure cache states (singleton sets) and one cache state that might contain sets with more than one element. This favors the idea to maintain as many pure cache states as possible. The complexity of the selection function Sel_1 is proportional to the number of cache states $O(Sel_1) = O(|C|)$.

A second selection strategy, Sel_2, is to choose from *all* cache states of C. This metric Sel_2 requires to compare all pairs of cache states leading to quadratic complexity $\mathcal{O}(Sel_2) = O(|C|^2)$. The metric Sel_2 is expected to yield more accurate results than Sel_1, because more elements are considered. An example for applying these metrics is given in Section 5.4.

5.2.3 *Merging Cache States.* The merge operation $\odot$ for two cache states is used in the algorithm $bound_Z$ in Figure 5, as well as in the iterative data flow algorithm for RCS and LCS calculation in Eq. (14).

We define the operation $\odot$ over M (M denotes the set of all memory blocks). It can also be used for cache states by applying it pointwise to its elements. Two cache states a and b with S cache sets can be merged by applying the union operator for sets to each element:

$$a \odot b = ((a_1[1] \cup b_1[1]), \cdots, (a_S[1] \cup a_S[1])) \tag{22}$$

The time complexity of the merge operation $\odot$ scales with the number of elements in each cache set X and the number of cache sets S of the cache:

$$O(a \odot b) = O(S \cdot (t_{\cup}(X, X)) = O(S \cdot X^2) \tag{23}$$

5.2.4 *Preemption Delay Calculation.* The cache-related preemption delay is computed by the intersection of useful cache blocks of the preempted task τ and the used cache blocks of the preempting task τ'. The same methodology of the approach by Mitra et al. can be applied. For a complete treatment, we specify the set of useful cache blocks:

$$USE^{\tau}_{scale}[B_i] = RCS^{\tau}_{out}[B_i] \cap LCB^{\tau}_{out}[B_i] \tag{24}$$

Finally, the preemption delay $CRPD^{\tau\tau'}_{scale}$ is given in Eq. (25) in which B_{end} denotes the last basic block of a task.

$$CRPD^{\tau\tau'}_{scale} = \max\{USE^{\tau}_{scale}[B_i] \cap RCS^{\tau'}_{out}[B_{end}] \quad | \quad \forall B_i \in \text{task } \tau\} \tag{25}$$

5.3 Time Complexity

The time complexity of the $bound_Z$ algorithm in Figure 5 is determined by the following steps.

1. complexity for counting number of cache states (line 1);
2. complexity for choosing c_i, c_j with $d(c_i, c_j)$ minimal (line 2);
3. complexity for removing and inserting a new cache state to C (line 3, 4);
4. complexity for merging two cache states c_i and c_j (line 4);
5. number of iterations of while-loop (line 1–5);

5.3.1 *Complexity for Each Step.* First, we describe the time complexity for each step within the while-loop, then we calculate the maximum number of while-loop iterations. All values of the time complexities are summarized in Table V.

Step 1 can be implemented in linear time relative to the number of elements of C. In the RCS (LCS) algorithm, as shown in Eqs. (12– 16), $|C|$ is bounded by the number of predecessor (successor) nodes. The maximum number of predecessor nodes can be bounded by 2, because `if-then-else` statements and loop-constructs like `for, while` create, at most, two branches. A `switch-case` statement, even though potentially many branches are created, can be represented by multiple `if-then-else` statements. Therefore, we can bound $|C| < 2Z$

Table V. Complexity of Analysis Steps for $Bound_Z$ Algorithm for Direct-Mapped Instruction Cache[a]

Step	Operation	Complexity
1	count	$O(2Z)$
2	Sel_1	$O(2Z)$
2	Sel_2	$O((2Z)^2)$
2	d_1	$O(S \cdot X)$
2	d_2	$O(S \cdot X^2)$
3	insert	$O(2Z)$
3	remove	$O(2Z)$
4	union	$O(S \cdot X^2))$
5	loop bound	Z

[a] Z denotes the maximum number of cache states at a node; X denotes the maximum number of memory blocks of a cache set; S denotes the number of cache sets in a cache.

because the RCS (LCS) each predecessor (successor) node has, at most, Z cache states.

Step 2 is determined by the time complexity of the selection algorithm Sel_i multiplied by the time complexity of the distance metric d_i plus finding the minimum element. The time complexity for each algorithm is summarized in Table V. Then, the smallest element can be found in $O(2Z)$ or $O((2Z)^2)$, depending on the selection method (Sel_1 or Sel_2).

Step 3 is determined by inserting and removing an element from a set. This can be done in linear time. Assuming that there are, at most, $2Z$ elements in set C, we get: $O(2Z)$. Step 4 is determined by the complexity for taking the union of each cache set with X elements: $O(S \cdot X^2))$.

Finally, we bound the maximum number of while-loop iterations. Since $|C| < 2Z$, there are, at most, Z loop iterations, because in each iteration the number of elements in C decreases by one element and the loop terminates when $|C| \leq Z$.

5.3.2 *Overall Time Complexity.* In summary, the complexity of the entire $bound_Z$ algorithm is given by multiplying the sum of counting number of elements, choosing the element with the minimum distance, removing and inserting an element, and taking the union of two sets with the maximum number of loop iterations:

$$O(bound_Z) = O(Z \cdot O(2Z) + O(Sel_i) \cdot O(d_i) + O(Sel_i) + 2O(Z) + O(S \cdot X^2))) \quad (26)$$

For distance metric d_1 and selection metric Sel_1 ,the complexity of the $bound_Z$ algorithm evaluates to:

$$O(bound_Z(d_1, Sel_1)) = O(Z^2 \cdot X + Z \cdot X^2)) \quad (27)$$

For the configuration of the distance and selection metric d_2 and Sel_2, the complexity evaluates to:

$$O(bound_Z(d_2, Sel_2)) = O(Z^3 \cdot X^2) \quad (28)$$

To conclude, the complexity scales quadratically with the number of cache states Z with the configuration d_1, Sel_1 and cubicly with Z for the configuration d_2, Sel_2 when we assume that X is a small constant. We will argue below that X is a small constant for most embedded applications.

5.3.3 *Bounding Number of Memory Blocks.* To tightly bound the maximum number of memory blocks X in a cache set s_i is difficult. A naive upper bound would be the number of memory blocks that map to cache set s_i, which is bounded by the total number of memory blocks M divided by the number of cache sets S: $\frac{M}{S}$. However, this is obviously an overestimation because a single access in a basic block to cache set s_i replaces all memory blocks of s_i by a single memory block, namely $gen[B]$. In regular frequency, the number of cache blocks X will be reduced to one.

When does the number of memory blocks increase? Only if the cache sets $s_i(p_1)$ and $s_i(p_2)$ in *both* predecessor nodes p_1 and p_2 of a basic block B are not empty and if no cache blocks are mapped to s_i during the execution of basic block B. The worst-case number of memory blocks in s_i occurs *after* the deepest nested `if-then-else` structure, with a full tree (each `then` and `else` branch exists), and a memory block is accessed in *each* branch at the lowest nested level. Assuming that d denotes the highest level of nested branch statements, then there can be 2^d memory blocks in s_i at all successor nodes b_k (after the deepest nest level) that do not access s_i, e.g., all b_k with $gen_{b_k} = \emptyset$. Then, X is bounded by in Eq. (29):

$$X \leq \min\left(2^d, \frac{M}{S}\right) \tag{29}$$

For time complexity considerations, we have to assume that the maximum number of elements occurs at each node. However, even if the maximum number of 2^d memory blocks is reached in a program, it will only be valid for a small number of nodes: e.g., the longest path where no accesses occur to cache set s_i. However, it is very difficult to determine formally how many basic blocks lie on that path.

As an alternative, we give some heuristic argument: The instruction cache size is significantly smaller than the number of all memory blocks (factor 10 to 100). There will be frequent accesses to the same cache set s_i, which reduces the total number of elements in a cache set to 1. It is, therefore, likely that the average number of elements in a cache set is small. In Section 8.3, we evaluate the average and maximum number of memory blocks for several benchmarks.

5.4 Example

We apply the scalable precision cache analysis to the example control-flow graph in Figure 2 and bound the number of cache states to $Z = 2$. First, we calculate the sets of RCS and LCS and, then, we compare the number of useful cache blocks to the results in the approaches by Lee and Mitra.

5.4.1 *Calculation of Reaching Cache States.* Table VI shows the result of the $RCS_{out}[B_i]$ for each node of the forward iterative data flow analysis of Eqs. (12–14). The result after the fixed point has reached is shown in Figure 6.

Table VI. Reaching Cache States $RCS_{out}[B_i\}]$ for Scalable Precision Cache Analysis with $Z = 2$

B_i	$RCS_{out}(B_i)$ 1st iter.	$RCS_{out}(B_i)$ 2nd iter.
B_1	[{0, ∅, ∅, ∅]	[{0}, {1, 5}, {6}, {11}] [{0}, {9}, {10}, {11}]
B_2	[{0}, {1}, {2}, {3}]	[{0}, {1}, {2}, {3}]
B_3	[{4}, {5}, ∅, ∅]	[{4}, {5}, {6}, {11}] [{4}, {5}, {10}, {11}]
B_4	[{0}, {1}, {6}, {3}] [{4}, {5}, {6}, ∅]	[{0}, {1}, {**6**}, {3}] [{4}, {5}, {**6**}, {11}]
B_5	[{0}, {1}, {6}, {7}] [{4}, {5}, {6}, {7}]	[{0}, {1}, {6}, {7}] [{4}, {5}, {6}, {7}]
B_6	[{8}, {9}, {10}, {3}]	[{8}, {9}, {10}, {3}] [{8}, {9}, {10}, {11}]
B_7	[{8}, {9}, {10}, {**11**}] [{0, 4}, {1, 5}, {6}, {**11**}]	[{0, 4}, {1, 5}, {6}, {11}] [{8}, {9}, {10}, {11}]

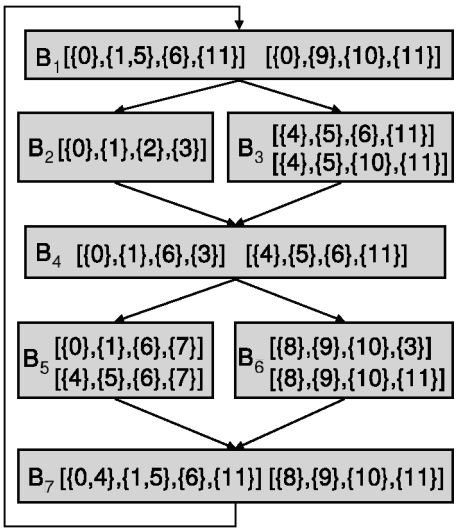

Fig. 6. RCS calculation for scalable precision cache analysis.

Two iterations are sufficient until the algorithm converges in this example. Initially, all $RCS_{in}[B_i] = \emptyset$. The $gen[B_i]$ for the control-flow graph in Figure 2 have been shown in Figure 1.

The *RCS* of the second iteration are shown graphically in Figure 6. We apply the scalable data flow analysis to this example, starting at B_1 in the first iteration. Because the *in* set is empty, $RCS_{out}[B_1] = gen[B_1]$. For basic block B_2, the incoming edge from B_0 is evaluated and Eq. (13) is applied. The result is shown in the second column in line 2 in Table VI. Analogously, all other $RCS_{out}[B_i]$ are computed for B_3, B_4, B_5, and B_6. The calculation at B_7 is different, because via the two incoming edges from B_5 and B_6, there are three cache states, but only $Z = 2$ states are allowed and, thus, the $bound_2$ algorithm is applied:

$$RCS_{in}[B_7] \;=\; bound_2([c_1 = \{0\}, \{1\}, \{6\}, \{7\}], c_2 = [\{4\}, \{5\}, \{6\}, \{7\}],$$

Table VII. Live Cache States $LCS_{in}[B_i]$ and $LCS_{out}[B_i]$ for Scalable Data Flow Analysis with $Z = 2$

B_i	$LCS_{in}[B_i]$	$LCS_{out}[B_i]$
B_1	[{0}, {1}, {2}, {3}] [{0}, {5}, {6}, {7, 11}]	[{0, 8}, {1}, {2}, {3}] [{4}, {5}, {6}, {7, 11}]
B_2	[{0}, {1}, {2}, {3}] [{8}, {1}, {2}, {3}]	[{0}, {1, 5}, {6}, {7}] [{8}, {9}, {6}, {11}]
B_3	[{4}, {5}, {6}, {7}] [{4}, {5}, {6}, {11}]	[{0}, {1, 5}, {6}, {7}] [{8}, {9}, {6}, {11}]
B_4	[{0}, {1, 5}, {6}, {7}] [{8}, {9}, {6}, {11}]	[{0}, {1, 5}, {6}, {7}] [{8}, {9}, {10}, {11}]
B_5	[{0}, {1}, {6}, {7}] [{0}, {5}, {6}, {7}]	[{0}, {1}, {2}, {11}] [{0}, {5}, {6}, {11}]
B_6	[{8}, {9}, {10}, {11}]	[{0}, {1}, {2}, {11}] [{0}, {5}, {6}, {11}]
B_7	[{0}, {1}, {2}, {11}] [{0}, {5}, {6}, {11}]	[{0}, {1}, {2}, {3}] [{0}, {5}, {6}, {7, 11}]

$$c_3 = [\{8\}, \{9\}, \{10\}, \{3\}])$$
$$RCS_{in}[B_7] = \{[\{0, 4\}, \{1, 5\}, \{6\}, \{7\}][\{8\}, \{9\}, \{10\}, \{3\}]\}$$

We use Sel_2 and d_1 as selection method and distance metric throughout this example:

$$d_1(c_1, c_2) = 2 \quad d_1(c_1, c_3) = 4, \quad d_1(c_2, c_3) = 4$$

Therefore, c_1 and c_2 are merged because their distance is the smallest (2). Note that the cache set $RCS_{in}[B_7][0]$ and $RCS_{in}[B_7][1]$ contain two memory blocks. Since the $gen[B_7]$ contains m_{11} in the last cache set, only the memory blocks in the last cache set are replaced. The $RCS_{out}[B_7]$ is shown in Table VI in the last line of the second column (in bold face).

The calculation in the second iteration of $RCS_{out}[B_i]$ is straightforward for B_1, B_2, and B_3. At B_4 cache states have to be merged, because there are three states available via incoming edges from B_2 and B_3:

$$RCS_{in}[B_4] = bound_2[c_1 = [\{0\}, \{1\}, \{2\}, \{3\}]c_2 = [\{4\}, \{5\}, \{6\}, \{11\}],$$
$$c_3 = [\{4\}, \{5\}, \{10\}, \{11\}]$$
$$RCS_{in}[B_4] = \{[\{0\}, \{1\}, \{2\}, \{3\}], [\{4\}, \{5\}, \{6, 10\}, \{11\}]\} \tag{30}$$

Again, we use Sel_2 and d_1: $d_1(c_1, c_2) = 4$, $d_1(c_1, c_3) = 4$ and $d_1(c_2, c_3) = 1$. The cache states c_2 and c_3 are merged, because they result in minimum distance of 1. The result is shown in Eq. (30). The $RCS_{out}[B_4]$ is shown in Table VI in the third column. We point out the replacement by $gen[B_4] = \{6\}$ (set in bold face). The analysis continues with B_5 and B_6 and, finally, the last merge-operation occurs at B_7. The four incoming states from B_5 and B_6 are evaluated using Sel_2 and d_1 metric. The result is shown in the last line in the second column in Table VI.

5.4.2 *Calculation of Live Cache States.* The results for the LCS_{in} and LCS_{out} of the scalable data flow algorithm are shown in Table VII. The LCS

Table VIII. Useful Cache Blocks USE_{scale} for Scalable Analysis with $Z = 2^a$

B_i	USE_{scale}	USE_{mitra}	USE_{lee}
B_1	**3**	3	4
B_2	2	2	2
B_3	**2**	2	3
B_4	**3**	3	4
B_5	**2**	2	3
B_6	1	1	1
B_7	**4**	3	4

[a] Comparison with state-based approach USE_{mitra} and set-based approach USE_{lee} from Table IV.

calculation uses a backward data flow analysis and has been given in the Eqs. (16) and (17).

Merge operations were necessary at B_1 in the first iteration and at B_4 and B_1 in the second iteration. Note the difference between $LCS_{out}[B_i]$ and $LCS_{in}[B_i]$. The former represents the live cache states *before* the memory blocks of B_i are loaded to the cache and the latter represents the live cache states *after* the memory blocks of B_i are loaded to the cache.

5.4.3 *Calculation of Useful Cache Blocks.* The useful cache blocks $USE_{scale}[B_i]$ are computed by the intersection of reaching cache states $RCS_{out}[B_i]$ and live cache states $LCS_{out}[B_i]$. The total number of useful cache blocks according to Eq. (24) are summarized and compared to the results of Lee and Mitra's approach in Table VIII.

The result is, in most cases, equal to the tighter result of USE_{mitra} and in one case equal to the result of USE_{lee} (at basic block B_7).

6. PREEMPTION DELAY ANALYSIS FOR ASSOCIATIVE CACHES

The general description in Section 5 considered only direct-mapped caches. This section extends the proposed analysis to set-associative caches. In an n-way associative cache, a memory block can be placed into n 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 when 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 a commonly used strategy.

Associative caches are commonplace in embedded architectures, but have mainly been ignored in preemption delay analysis. The work described in Mitra et al. [2003], Petters and Färber [2001], and Busquets-Mataix et al. [2000] target direct-mapped instruction caches. Only in Lee et al. [1998] have associative caches been considered. However, their description contains a flaw, which we point out in Section 6.6. In the following, we present the analysis for associative instruction caches with *LRU* replacement strategy based on the new scalable precision cache model.

6.1 Scalable Data Flow Analysis

According to our definition in Section 5, 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 model has to be extended. In the following, we formulate the computation of reaching cache states RCS. 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 $gen^c[B]$ represents a vector of n sets of memory blocks:

$$gen^c[B] = (gen^c_1[B], gen^c_2[B], \cdots, gen^c_n[B]) \tag{31}$$

The $gen^c[B]$ set contains the *last* memory blocks that are accessed during the execution of basic block B. More formally, each $gen^c_i[B]$ is either empty or $gen^c_i[B] = \{m\}$ if m is the ith most recently accessed memory block in B. The set $gen^s_n[B]$ contains the most recently accessed cache block and $gen^s_1[B]$ the least recently one. With this definition of $gen^c[B]$, the sets $RCS^c_{in}[B]$ and $RCS^c_{out}[B]$, are related as follows:

$$RCS_{in}[B] = bound_Z\left(\bigcup_{p\in pred(B)} RCS_{out}[p]\right) \tag{32}$$

$$RCS^c_{out}[B] = \bigcup_{r\in RCS^c_{in}[B]} LRU_{gen^c_1}\big(\cdots\big(\big(LRU_{gen_n}(r)\big)\big)\big) \quad \forall c.1 \le c \le S \tag{33}$$

The function $bound_Z(C)$ is the same as in Figure 5. The replacement algorithm for the scalable precision cache model $LRU_m(c)$ for an n-way associative cache is presented in Section 6.3. Note that the $RCS_{in}[B]$ is defined for each cache state and $RCS^c_{out}[B]$ is defined for each cache set c. However, this is only a matter of presentation.

6.2 Bound Algorithm for Associative Caches

In the following, we extend the distance metric, selection algorithm, and merge operation for associative caches, based on the presentation in Section 5.2 for the $bound_Z$ algorithm.

6.2.1 *Distance Metric.* The function $d(a, b)$ of two cache states a, b is defined as a metric that delivers the difference of two cache states. The metrics d_1 and d_2 are applied to n-way associative caches as follows:

$$d_1(a,b) = \sum_{s=1}^{S} \begin{cases} 1 & if\ \exists\ i.\quad a_s[i] \neq b_s[i] \quad \forall \le i \le m \\ 0 & otherwise \end{cases} \tag{34}$$

The complexity of metric d_1 is

$$O(d_1) = O(S \cdot m \cdot X) \tag{35}$$

where X denotes the maximum number of elements of the set $a_s[i]$. Equality test of two sets can be computed in linear time [Cormen et al. 1997]. A bound for the maximum number of memory blocks X is discussed in Section 5.3.3.

```
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
```

Fig. 7. LRU algorithm for scalable cache model.

The metric d_2 counts the number of different memory blocks of each cache set with the symmetric difference:

$$d_2(a,b) = \sum_{s=1}^{S} \sum_{nn=1}^{n} \left| (a_s[nn] \cup b_s[nn]) \setminus (a_s[nn] \cap b_s[nn]) \right| \tag{36}$$

The complexity of d_2 is given by taking $S \cdot n$ times the following operations: (1) the union of two sets with totally $2X$ elements; (2) the intersection of two sets with at most $2X$ elements; and (3) the set difference of a set with $2X$ elements and X elements.

$$O(d_2) = O(S \cdot m \cdot (t_\cup(X,X) + t_\cap(X,X) + t_{setdiff}(2X,X)) = O(S \cdot m \cdot X^2)) \tag{37}$$

6.2.2 *Cache State Selection.* The selection methods Sel_1 and Sel_2 are independent of the associativity. Therefore, the complexity is the same as in Section 5.2: $O(Sel_1) = O(|C|)$ and $O(Sel_1) = O(|C|^2)$.

6.2.3 *Merging of Cache States.* The merge operation $\odot$ has already been defined for direct-mapped caches in Section 5.2. For an n-way associative cache, the $\odot$ operation is applied pointwise to every vector element $a_i[1], \cdots, a_i[n]$ of a cache set a_i. The time complexity is given by:

$$O(a \odot b) = O(S \cdot n \cdot (t_\cup(X,X)) = O(S \cdot n \cdot X^2)) \tag{38}$$

6.3 LRU Algorithm for Scalable Precision Cache Model

The replacement algorithm $LRU_{gen_i^c}(r)$ for cache set c of cache state r is defined in the following Eq. (39):

$$LRU_{gen_i^c}(r) = \begin{cases} r & if \quad gen_i^c = \emptyset \\ LRU_m(r) & if \quad gen_i^c = \{m\} \end{cases} \tag{39}$$

The function $LRU_m(r)$ models the LRU replacement strategy for the scalable precision cache model. The pseudocode is shown in Figure 7.

LEMMA. *The algorithm in Figure 7 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 last 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 model stepwise 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 an 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 7.
(b) $\exists c_i. \quad m \in c_i$. From the assumption $|c_i| \leq 1$ it 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 to $n-1$ 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 (Part I.)
(b) There exists a unique $c_i. \quad m \in c_i$. If $|c_i| \leq 1$, refer to Part I, otherwise the case $|c_i| = d \geq 1$ is detected in lines 10–11 in the algorithm. We have to distinguish two cases:

$$(\text{b1})(c_1, \cdots, c_{i-1}, \{m\}, c_{i+1}, \cdots, c_n)$$
$$(\text{b2})(c_1, \cdots, c_{i-1}, \{m_k\}, c_{i+1}, \cdots, c_n), \forall m_k \in c_i.m_k \neq m$$

For (b1) we have shown already in Part 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 an 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'. Let us 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. Induction step: 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

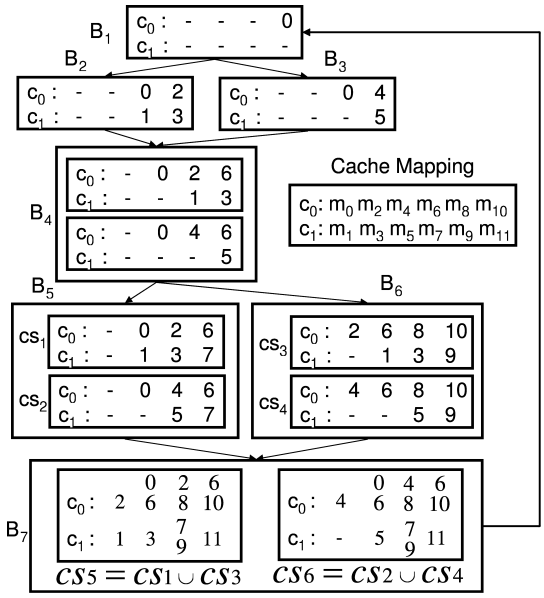

Fig. 8. Reaching cache states for four-way associative instruction cache with two sets.

do not contain m are singleton sets. If a c_i contains more than one element, we construct every possible cache state and apply Part II for each state; the union is then taken (line 10–11).

This completes the proof. □

6.4 Example

We apply the algorithm to the control flow graph of Figure 2 with a four-way associative instruction cache with two cache sets. For demonstration, we compute the reaching cache states (RCS) for each node. Figure 8 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 and an empty set is denoted with $-$. In order to demonstrate the cache state reduction, we restrict the number of cache states to $Z = 2$. In B_4, two cache states are reached and only the memory block m_6 is mapped to c_0 ($gen^{c_0}[B_4] = [\{\}, \{\}, \{\}, \{m_6\}]$). Since $m_6 \notin c_0$ in both cache states, the condition in line 2 is true and the lines 3–4 of LRU algorithm in Figure 7 are executed. All memory blocks move one position to the left and m_6 is placed 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 $RCS_{in}[B_7]$, according to Eq. (32), four cache states cs_1, cs_2 cs_3, and cs_4 are available on incoming edges, but only two are allowed. Therefore, two of the cache states with the minimum distance, according to Eq. (20), will be merged. We assume, for this example, the selection

method Sel_2, which compares all cache states, and distance metric d_2, which computes the symmetric difference.

$$d_2(cs_1, cs_2) = 5 \quad d_2(cs_1, cs_3) = 5 \quad d_2(cs_2, cs_3) = 10$$
$$d_2(cs_1, cs_4) = 10 \quad d_2(cs_2, cs_4) = 5 \quad d_2(cs_3, cs_4) = 5$$

Therefore, cache states $cs_1 \cup cs_3$ and $cs_2 \cup cs_4$ are merged:

$$cs_5'' = cs_1 \cup cs_3 = \begin{cases} c_0 & [\{2\}, \{0, 6\}, \{2, 8\}, \{6, 10\}] \\ c_1 & [\emptyset, \{1\}, \{3\}, \{7, 9\}] \end{cases}$$
$$cs_6'' = cs_2 \cup cs_4 = \begin{cases} c_0 & [\{4\}, \{0, 6\}, \{4, 8\}, \{6, 10\}] \\ c_1 & [\emptyset, \emptyset, \{5\}, \{7, 9\}] \end{cases}$$

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. To save, space, several elements within a set at a position are aligned vertically, such as m_6 and m_{10} in cache set c_0.

In the second iteration of the data flow analysis, we start again with basic block B_1. Now $m_0 \in c_0$ in both cache states c_5'' and c_6''. Therefore, the loop in line 7 is executed once, the contents of cache set $c_0[2]$ to $c_0[4]$ are moved one position to the left and m_0 is placed in the $c_0[4]$ slot. This results to the new cache states cs_5 and cs_6:

$$cs_5 = \mathrm{LRU}_{m_0}(cs_5') = \begin{cases} c_0 & [\{\mathbf{2}, \mathbf{6}\}, \{\mathbf{2}, \mathbf{8}\}, \{\mathbf{6}, \mathbf{10}\}, \{\mathbf{0}\}] \\ c_1 & [\{1\}, \{3\}, \{7, 9\}, \{11\}] \end{cases}$$
$$cs_6 = \mathrm{LRU}_{m_0}(cs_6') = \begin{cases} c_0 & [\{\mathbf{4}, \mathbf{6}\}, \{\mathbf{4}, \mathbf{8}\}, \{\mathbf{6}, \mathbf{10}\}\{\mathbf{0}\}] \\ c_1 & [\emptyset, \emptyset, 5, \{7, 9\}] \end{cases}$$

As the last example of the algorithm, we show the the RCS computation for the cache state cs_5 at basic block B_2 with $gen^{c_0}[B_2] = [\emptyset, \emptyset, \emptyset, \{m_2\}]$ and $gen^{c_1}[B_2] = [\emptyset, \emptyset, \{m_1\}, \{m_3\}]$.

Now the LRU algorithm is applied three times:

$$RCS_{out}[B_2] = LRU_{m_3}(LRU_{m_2}(LRU_{m_1}(cs_5)))$$

We show the computation in three steps and denote each intermediate cache state as cs', cs'', and cs''' respectively:

$$cs' = LRU_{m_1}(cs_5) = \begin{cases} c_0 & [\{2, 6\}, \{2, 8\}, \{6, 10\}, \{0\}] \\ c_1 & [\{\mathbf{3}\}, \{\mathbf{7}, \mathbf{9}\}, \{\mathbf{11}\}, \{\mathbf{1}\}] \end{cases} \tag{40}$$

$$cs'' = LRU_{m_2}(cs') = \begin{cases} c_0 & [\{\mathbf{6}, \mathbf{8}\}, \{\mathbf{6}, \mathbf{10}\}, \{\mathbf{0}\}, \{\mathbf{2}\}] \\ c_1 & [\{3\}, \{7, 9\}, \{11\}, \{1\}] \end{cases} \tag{41}$$

$$cs''' = LRU_{m_3}(cs'') = \begin{cases} c_0 & [\{6, 8\}, \{6, 10\}, \{0\}, \{2\}] \\ c_1 & [\{\mathbf{7}, \mathbf{9}\}, \{\mathbf{11}\}, \{\mathbf{1}\}, \{\mathbf{3}\}] \end{cases} \tag{42}$$

The access of $LRU_{m_1}(cs_5)$ shows a reordering in cache set c_1, where m_1 is placed in the last recently used position $c_1[3]$. The result is shown in Eq. (40).

The access of $LRU_{m_2}(cs')$ is shown in Eq. (41), which is somewhat more difficult to explain, because $m_2 \in c_1[1]$ and $m_2 \in c_1[2]$. We explain the algorithm by

Table IX. Complexity of Analysis Steps for $Bound_Z$ Algorithm for n-Way Associative Instruction Caches[a]

Step	Operation	Complexity
1	count	$O(2Z)$
2	Sel_1	$O(2Z)$
2	Sel_2	$O((2Z)^2)$
2	d_1	$O(S \cdot n \cdot X)$
2	d_2	$O(S \cdot n \cdot X^2)$
3	insert	$O(Z)$
3	remove	$O(Z)$
4	union	$O(S \cdot n \cdot X^2))$
5	loop bound	Z

[a] Z denotes the maximum number of cache states at a node, X denotes the maximum number of memory blocks of a cache set, and S denotes the number of cache sets in a cache.

expanding all possible cache states that are represented by cs', then we apply the LRU algorithm and, finally, we compare the result with the LRU algorithm for the scalable precision model. We leave out all invalid states for cache set c_0: [2, 2, 6, 0], [2, 2, 10, 0], [6, 2, 6, 0], and [6, 8, 6, 0], because a memory block m_i occurs several times. If we apply the standard LRU algorithm to the remaining cache states, we get:

$$LRU_{m_2}[2, 8, 6, 0] = [8, 6, 0, 2] \quad LRU_{m_2}[2, 8, 10, 0] = [8, 10, 0, 2]$$

$$LRU_{m_2}[6, 8, 10, 0] = [8, 10, 0, 2] \quad LRU_{m_2}[6, 2, 10, 0] = [6, 10, 0, 2]$$

The union of the above resulting cache states is then the conservative cache state cs'' that is computed by the algorithm in Figure 7 (lines 6–12).

Finally $LRU_{m_3}(cs'')$ is applied in Eq. (42), which is a reordering for cache set c_1.

Note, that the set-based cache model may lead 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.

6.5 Time Complexity

The time complexity for n-way associative caches is similar to the complexity discussion in Section 5.3.

The complexity of the $bound_Z$ algorithm as shown in Figure 5, depends on the same steps. The complexity of each step is summarized in Table IX.

The time complexity for the LRU algorithm is calculated easily, examining each line of the algorithm in Figure 7:

$$\begin{aligned} O(LRU_m(r)) &= \max(1 + X \cdot n, 1 + 2t_{\cup}(X, X) \cdot n + X + nt_{rem}) \\ O(LRU_m(r)) &= 2O(X^2) \cdot n + O(X)(n + 1) + 1 = (C^2 \cdot n) \end{aligned} \tag{43}$$

To conclude, the LRU algorithm scales quadratically with the number of cache states C times the degree of associativity.

In summary, the complexity of the entire $bound_Z$ algorithm is given by

$$O(bound_Z) = O(Z \cdot (O(Sel_i) \cdot O(d_i) + O(Sel_i) + 2O(2Z) + O(cs \cdot m \cdot X^2))) \quad (44)$$

For the distance and selection metric d_1 Sel_1 the complexity of the $bound_Z$ algorithm evaluates to:

$$O(bound_Z) = O(Z^2 \cdot X + Z \cdot X^2)) \quad (45)$$

and the complexity for the distance and selection metrics d_2 and Sel_2 evaluates to:

$$O(bound_Z) = O(Z^3 \cdot X^2) \quad (46)$$

To conclude, the complexity scales quadratically with the number of cache states with the configuration d_1, Sel_1 and cubicly with d_2, Sel_2 when we assume that X is a small constant. This is the same time complexity as for direct-mapped caches.

6.6 Lee's Approach Contains a Flaw

The approach by Lee et al. [1998] extends the data flow analysis to set-associative caches. The proposed solution is principally correct, but contains a flaw, which we would like to point out. If a basic block accesses a memory block that already exists in the RMB_{in} set, then this cache block can occur multiple times in the proposed algorithm by Lee et al. [1998] (Section 7.1). The correct solution would be to reorder the memory blocks. In Mueller [2000], the program line reordering for LRU has been considered for cache analysis for instruction caches. In the following, we briefly review the algorithm by Lee et al. [1998] and give an example.

The reaching cache states are denoted by $RMB^c_{in}[B]$ and $RMB^c_{out}[B]$ for the incoming and outgoing reaching cache blocks of cache set c at basic block B. A state of a cache set for an n-way set-associative cache is defined as a vector $(m_{i_1}, \cdots, m_{i_n})$, where m_{i_1} is the least recently referenced block and m_{i_n} the most recently referenced block. The $gen^c[B]$ contains the state of cache set c generated in basic block B. It is either empty when none of the memory blocks mapped to cache set c are referenced in basic block B or a singleton set whose only element is a vector: $(gen^c_1[B], \cdots, gen^c_n[B])$. In the vector $gen^c_n[B]$, is the memory block whose reference in basic block B is the last reference to the cache set c in B.

We give the an example for a four-way associative cache with the following assumptions about $RMB^c_{in}[B]$ and $gen^c[B]$ at some basic block B in cache set c:

$$RMB^c_{in}[B] = (m_1, m_2, m_3, m_4) \quad gen^c[B] = (null, null, m_3, m_5)$$

Note, that m_3 is reached via some incoming path *and* is referenced in basic block B. The $RMB^c_{out}[B]$ is defined in Lee's algorithm depending on the contents of the *gen*-set. We give the data flow equations for RMB_{in} and RMB_{out} only for the case that applies to this example:

$$RMB^c_{in}[B] = \bigcup_{p \in pred(B)} RMB_{out}[p]$$

$$RMB_{out}[B] = \begin{cases} \bigcup_{rmb \in RMB^c_{in}[B]} \{(rmb_{n-1}, rmb_n, gen^c_{n_1}[B], gen^c_n[B])\} \\ \quad \text{if } gen^c_1[B], \cdots gen^c_{n-2}[B] = null \text{ and } gen^c_{n-1}[B], gen^c_n[B] \\ \quad \neq null \\ \cdots \text{ otherwise} \end{cases} \tag{47}$$

For the above example, the $RMB^c_{out}[B]$ is given by

$$RMB^c_{out}[B] = (rmb_3, rmb_4, gen^c_3[B], gen^c_4[B]) = (\mathbf{m_3}, m_4, m_3, m_5)$$

However, when the memory blocks $m_3 \rightarrow m_5$ are loaded to cache set c, the LRU replacement algorithm will result as:

$$(\mathbf{m_2}, m_4, m_3, m_5)$$

The mistake occurs for memory block m_3, which is reordered by the LRU algorithm. In the algorithm by Lee, the *gen*-sets and RMB_{in} sets are not examined for equal elements. Therefore, m_3 occurs in $RMB^c_{out}[B]$ multiple times and m_2 is missing, which might result in an underestimation of the CRPD.

7. CACHE-ANALYSIS FRAMEWORK FOR REAL-TIME VERIFICATION

The verification of real-time behavior involves the computation of worst-case response times. Typically, several embedded applications run on an embedded micro controller with an real-time operating system, such as ERCOSEK[1] for automotive applications. The response-time analysis has to be extended to consider the preemption delays because of cache reloads.

7.1 Cache-Aware Response-Time Analysis

Preemption delay analysis alone does not solve the cache-behavior problem. A cache-aware response-time analysis has to calculate how often a task is activated and how often a task is preempted for a given set of tasks.

As reviewed in Section 2, several cache-aware response-time approaches have been proposed [Busquets-Mataix and Wellings 1996; Petters and Färber 2001; Lee et al. 2001]. The approach by Busquets-Mataix and Wellings [1996] considers only the preempting task, while approach by Petters and Färber [2001] consider only the preempted task. However the advantage is that the time complexity is about the same as the time complexity for solving the fixed point of the underlying iterative response-time equations. In the approach by Lee et al. [2001] the preempted, as well as the preempting task, are considered, but the cache-aware response-time analysis scales exponentially with the number of tasks in the system.

In previous work, we have developed a cache-aware response-time analysis [Staschulat et al. 2005] for fixed priority preemptive scheduling. It computes the total number of preemptions in a given schedule in a polynomial time complexity. Therefore, it suits a framework with an overall low time complexity. The preemption delay is computed by a state-based data flow analysis [Staschulat

[1]Automotive Real-Time Operating System by ETAS.

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

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

and Ernst 2004]. It turned out that the complexity of the state-based approach was too large for greater sets of software tasks.

Therefore, we replace the time-consuming preemption delay analysis of Staschulat and Ernst [2004] by the scalable precision cache analysis as presented in this paper. In this paper, we apply the cache-aware response time analysis in several experiments to the results for the scalable precision cache analysis.

7.2 Pseudo-LRU Replacement Strategy

Sometimes four-way associative caches and above processors implement pseudo-LRU. As long as this replacement strategy is deterministic, the replacement algorithm in Figure 7 can be adapted accordingly.

7.3 Guidance to Choose Scaling Parameter

We give more guidance about how to choose the scaling parameter in the following. The maximum number of memory blocks within a cache set occurs in the deepest nested branch statement, as described in Section 5.3.3. This value 2^d, where d is the nest level, can be used as a reference. However, in reality, a much smaller number will suffice, because the nested branch is not a full tree and cache references do not occur to the same cache set in every leaf node of the branch tree.

8. EXPERIMENTS

8.1 Setup

This section presents the experimental results for the described analysis method for five benchmarks taken from Lee et al. [1998], Mitra et al. [2003], and Stappert [2003] and five preemption scenarios (PrS).

Table X summarizes the benchmark characteristics: main memory usage in bytes [B], the number of C source code lines, and the WCET in 10^3 clock cycles [clk] for a four-way set associative 1 KB instruction cache. The worst-case execution time of each task was determined with SymTA/P[2] using the cycle accurate ARM945 processor simulator[3] for the different instruction cache architectures with a 20-cycle cache-miss penalty. Each instruction is four bytes long and the cache block size is fixed to eight bytes, such that two instructions

[2]www.ida.ing.tu-bs.de/research/projects/symta.

[3]Realview development suite. www.arm.com.

Table XI. Cache Footprint Index for a Direct-Mapped Cache for Different Preemption Scenarios (PrS) and Cache Sizes

PrS	Preempting, Preempted Task	256 B	512 B	1024 B	2048 B
A	τ_1 τ_2	1.53	0.95	0.47	0.24
B	τ_1 τ_3	2.0	1.68	1.31	0.83
C	τ_1 τ_5	2.0	1.68	1.34	1.17
D	τ_4 τ_3	2.0	2.0	1.97	1.23
E	τ_4 τ_5	2.0	2.0	2.0	1.57

fit in a cache block. Since these benchmarks are rather small compared to a real application, the cache size has to be adjusted accordingly. However, if both the application size and cache size are considered using a cache footprint index, our results can be scaled to larger applications.

The cache footprint index determines how many tasks use a single cache block on average. Table XI shows this index for the evaluated preemption scenarios for a direct-mapped cache and varying cache sizes.

For example in the PrS B, both tasks τ_1 and τ_2 fully utilize the 256B cache. Hence, the footprint index is 2. The footprint of 1.17 for PrS C and 2KB cache shows that, on average, a cache block is used by one task only. For PrS A and 2 KB cache the footprint index of 0.24 represents a small application and a large instruction cache. The cache footprint index can be used to generalize the results of this paper for real-size applications, since the cache behavior depends on its utilization, and not just on cache size or application size alone.

8.2 Preemption Delay Analysis

We have implemented the analysis approach described in the preceding sections for direct-mapped and associative-instruction caches. We assume the distance d_2 metric of Eq. (20) and the strategy Sel_2, that compares all possible cache states.

The following diagrams show the analysis precision and analysis time compared to previously published approaches by other researchers. First, we compare the bound of the preemption delay, considering only the preempted, only the preempting and both (preempting as well as preempted) task. We then evaluate the influence of the scaling parameter (the number of cache states which are at most allowed at each node during data flow analysis) on analysis precision. We describe the effects for changing the cache size, associativity, and different benchmarks. Third, we show the impact of the scaling parameter on analysis time and memory consumption. Finally, we report the measured number of memory blocks of each set, as defined in Eq. (29), which influences the timing complexity of the intersection and merge operation (see Section 5.3.3).

The following results show the estimated preemption delay if a task preempts another task once. Figure 9 shows the number of *used cache blocks of the preempting task*, the number of *useful cache blocks of the preempted task*, and the cache-related preemption delay (*CRPD*), as computed in Eq. (25) for the preemption scenarios in Table XI. (See also Figure 1 and Section 5.2.4 for a graphical overview and the formulas.)

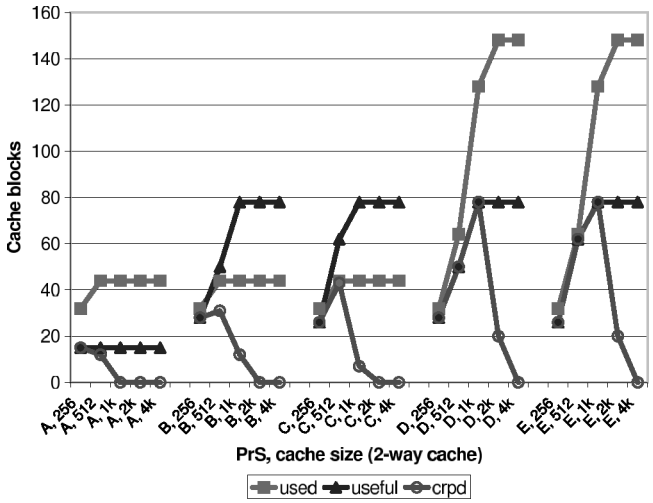

Fig. 9. Used, useful and CRPD estimation for 2-way associative cache.

The curves for used and useful cache blocks are very similar and its shape can be structured in several phases. In the first phase, the cache size is the limiting factor. The number of cache blocks (used, useful) proportionally grows with the number of cache blocks in the cache. In the second phase, all useful cache blocks fit in the cache and the curves remain on a constant level.

The *crpd* curve can be structured in three phases. First, the curve increases with the cache size, as for the *used* and *useful* curve, since the cache size is the limiting factor until the curve reaches a maximum. In the second phase, the *crpd* values are decreasing, because the *useful* and *used* cache blocks do not fully overlap (smaller cache foot print index). In the third phase, the *crpd* value is zero. This is because, the *used* and *useful* cache blocks of the preempting and preempted task, respectively, are mapped to different cache sets and do not overlap. Note also that to reach phase three for a given preemption scenario it is not necessary that both tasks *entirely* fit in the cache.

These results clearly show that just considering the *used* or just *useful* alone will lead to a pessimistic bound of the preemption delay.

As the next observation, we note that the *crpd* value is bounded by the minimum value of the *used* and *useful* curve. Its important to note, that the *useful* curve is not below the *used* curve, because they are computed for *different* tasks (for example, PrS B and C). Of course, $useful(\tau_i) \leq used(\tau_i)$ for the same task τ_i. Figure 10 shows the influence of the cache associativity. The *used* curve might increase with higher associativity, while the *useful* curve remains at the same level.

Figure 11 shows the bound of the preemption delay for a single preemption in number of cache blocks for increasing cache sizes. Observe the three phases of the *crpd* curve. In phase one, the value increases with the cache size, e.g., for 256 B most preemption scenarios have the same value (total number of cache blocks of the cache. In the second phase, the values decrease. In phase three, the preemption delay *crpd* is zero.

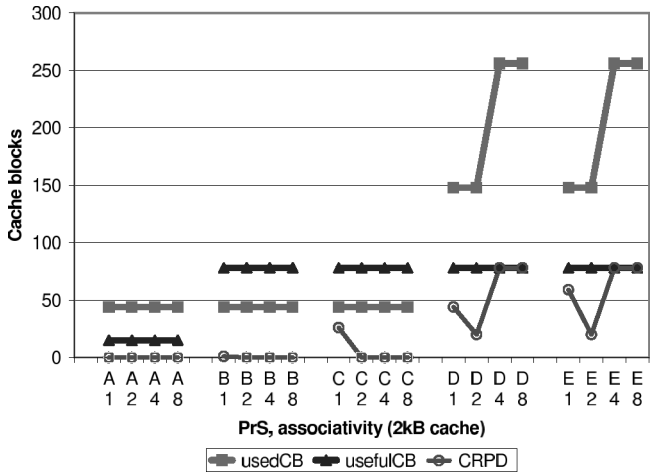

Fig. 10. Used, useful and CRPD estimation for 2kB cache.

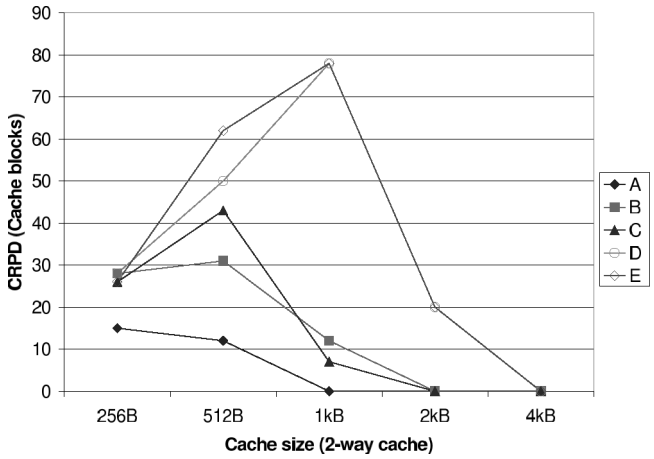

Fig. 11. Three phases of *crpd* curve.

The next Figures 12, 13, and 14 show the impact of the scaling parameter. Figure 12 shows the impact when $z = 1, 10, 20, \infty$ for different cache sizes for a two-way associative cache.

For $z = 1$ the results represent one cache state at each basic block as in approach by Lee et al. [2001]. The merge operation will operate exclusively on a single set.

For $z = \infty$, the results represent all possible cache states. The merge operation will never be applied. Thus, this modeling is equivalent to the approach by Mitra et al. [2003]. In all the other cases, the parameter z corresponds to the bounded-number cache states. Figure 12 shows, in most cases, no improvement of the *crpd* estimation.

For a higher associativity, Figure 13 shows an improvement at already a small number of states. For PrS D and E, no value for $z = \infty$ could be calculated because the memory consumption was too large (see later Figure 16). Note, the

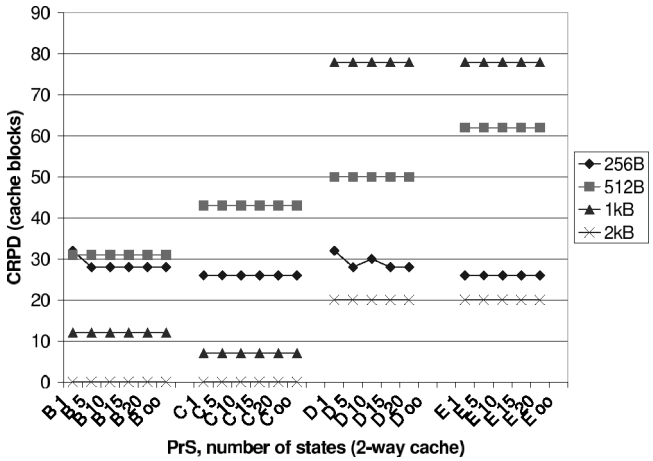

Fig. 12. Impact of number of cache states.

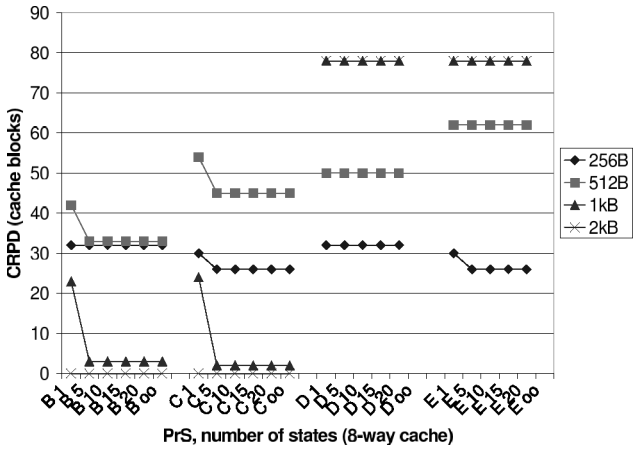

Fig. 13. Impact of cache size.

crpd-value for $z = 20$ is the same as for $z = \infty$ for all cases. Figure 14 shows that for increasing associativity, the analysis results are very similar.

8.3 Analysis Time and Memory Consumption

In this subsection, the performance of the analysis framework itself is evaluated in terms of analysis time, memory consumption, and other statistical measures. The analysis framework was executed on a 1.7 GHz machine with 4 GB RAM.

Figure 15 shows the analysis time (s) for a typical setup (e.g., one-way associative cache) for different cache sizes and preemption scenarios. Note the logarithmic scale. This highlights the exponential growth in running time (PrS C took several hours). Only for the smaller benchmark, could we execute the analysis for $z = \infty$; no results could be obtained for PrS D and E. For caches with a higher associativity, the analysis times are slightly higher (not shown in diagram).

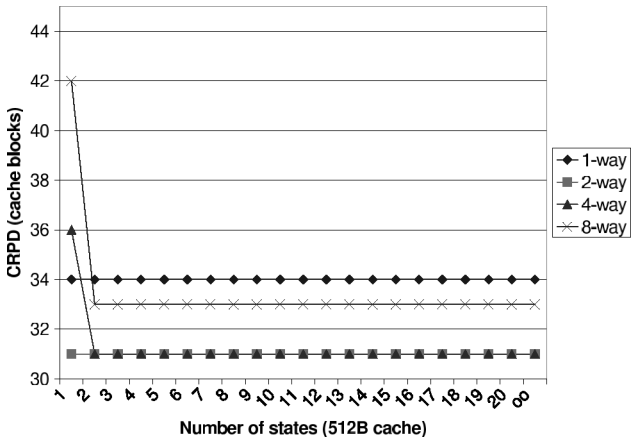

Fig. 14. Impact of associativity.

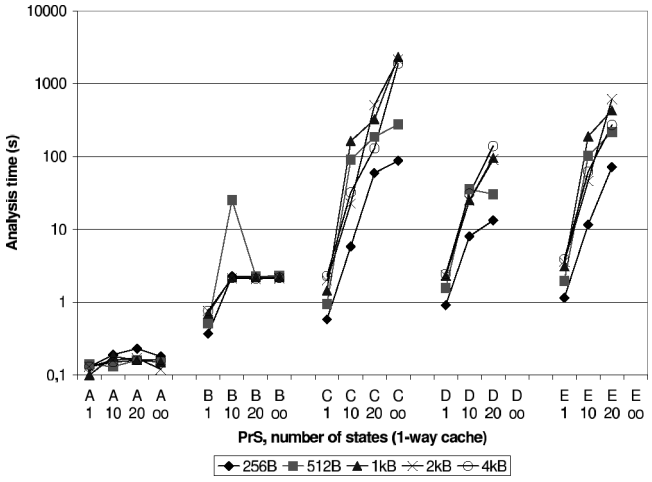

Fig. 15. Analysis time.

Figure 16 shows the memory consumption of the analysis framework for a typical setup. While the memory consumption is about 3 to 100 MB for $z = 1$ to $z = 20$, the memory consumption for the larger task τ_5 in PrS C is in the range of giga-bytes for $z = \infty$; for PrS D and PrS E no results have been obtained, because too many cache states were necessary.

Figures 17 and 18 show the maximum and average number of memory blocks in each set, as defined in Section 5.3.3. While there exists a higher number of memory lines (about 20 for PrS E), the average number of memory blocks is between one and two for all benchmarks and cache sizes. This has been checked for all setups, but cannot be shown here. These results give strong evidence that the number of X can be considered as a small constant for the timing complexity analysis, as suggested in Section 5.3.3.

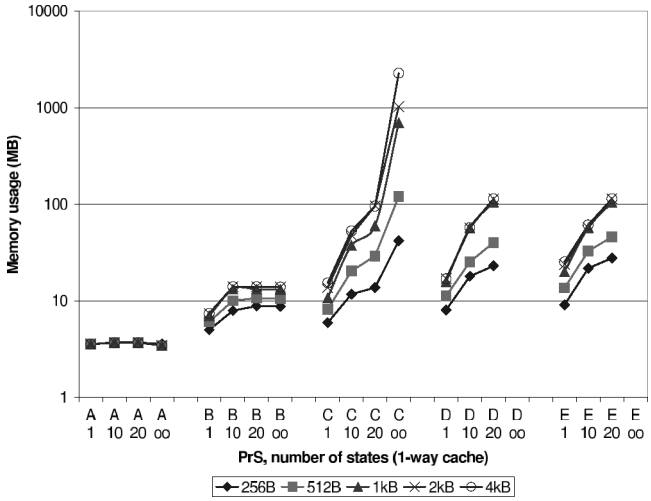

Fig. 16. Memory consumption.

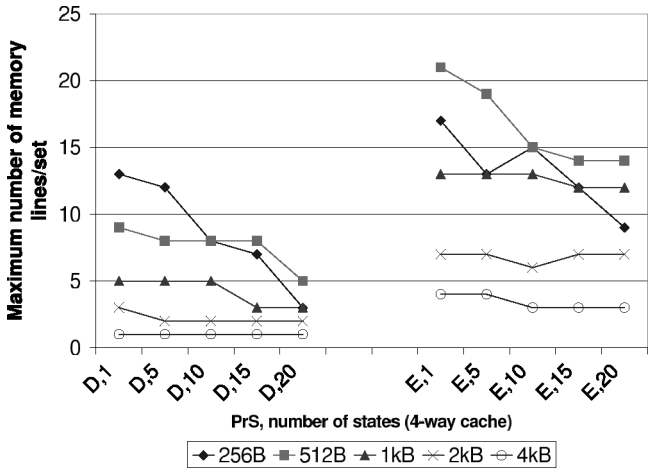

Fig. 17. Maximum number of memory blocks per set.

8.4 Response Time Analysis

Finally, we integrate the scalable CRPD estimation into a response-time analysis as proposed in Staschulat et al. [2005]. It reveals the impact of the cache-related preemption delay on the total response time of a task. Our approach is compared to the approaches by Busquets-Mataix et al. [Busquets-Mataix and Wellings 1996] and Petters [Petters and Färber 2001]. Busquets assumes that all used cache blocks of the preempting task are removed. However, in their experiments, they assume that the entire cache is flushed. In the following experiments, we use the actual number of used cache blocks of the preempting task, which is computed by the global data flow analysis. Petters' presentation is based on the number of useful cache blocks. However, the

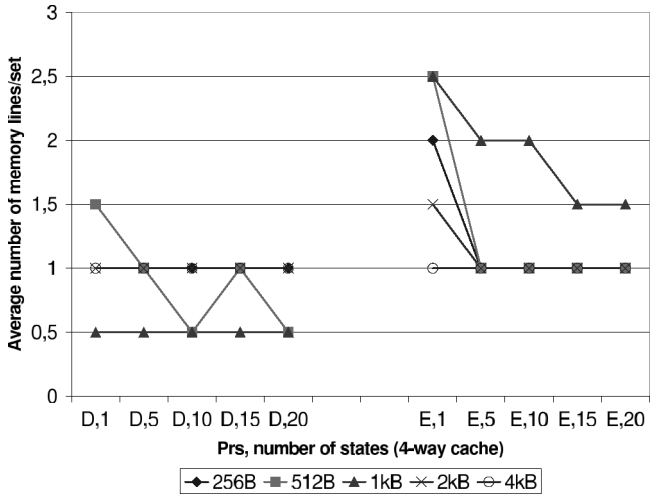

Fig. 18. Average number of memory blocks per set.

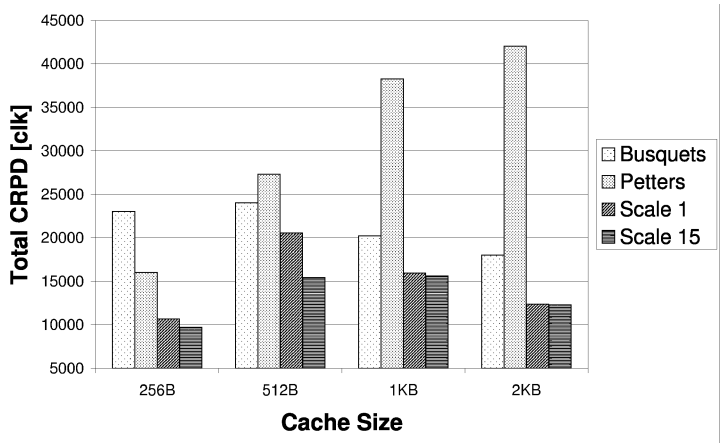

Fig. 19. CRPD in clock cycles (clk) during the entire response time of τ_4.

author only assumes a fixed percentage of cache content to be useful without performing any analysis. Again, because we have the total number of useful cache blocks available from global data flow analysis, we use this number in Petters' computation. The approach by Lee et al. [2001] needs an exponential number of equations for the ILP formulation and a reimplementation would have been too time consuming for comparison purposes.

We compute the worst-case response time for the task set τ_1, τ_2, τ_3, τ_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, which is why we left it out. Figure 19 shows the total preemption delay in clock cycles during the entire schedule for several cache sizes for a four-way set associative cache. Compared to Busquets, the data flow analysis with scaling factor 15 shows an improvement of 57, 35, 22, and 31% for 256 and 512 B, 1 KB and 2 KB cache

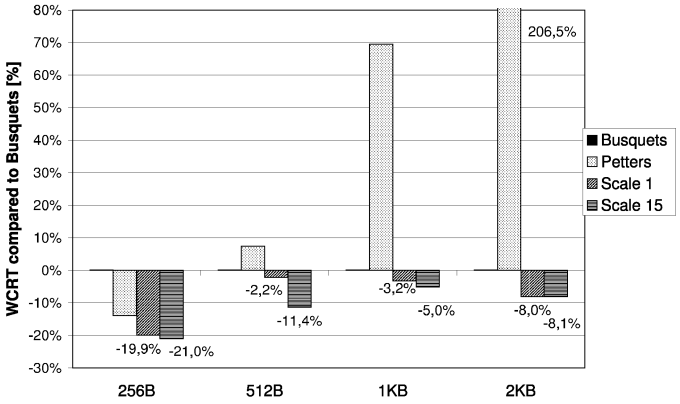

Fig. 20. CRPD and core execution time (response time) for τ_4.

respectively. Compared to Petters, our analysis shows an improvement of 39, 43, 59, and 70% for the given cache sizes.

The response time of task τ_4 is shown in Figure 20. This value includes the preemption delay (CRPD) of Figure 19, as well as the core execution times of each task and the time of higher priority tasks according to Eq. (3). The total preemption delay is calculated by the previous developed cache-aware response-time analysis [Staschulat et al. 2005].

The vertical scale shows the response time in percentage of the response time that was calculated by Busquets approach. The reason is that the response times for different cache sizes are significantly different, such that a linear scale would be inappropriate. The value for Busquets is zero, in all cases, because all other values were normalized to it. Nevertheless, we kept it in the figure to compare it to our results.

Besides Busquets, the results in Petters' approach, the set-based approach with scaling factor 1 and 15 are depicted. The analysis precision for scaling factor 15 improves between 5% for 1 KB cache and 21% for 256 B cache.

8.5 Discussion

From the experiments, we have gained the following insights:

- While the analysis precision is comparable to the state-based approach, the time and memory consumption is directly controlled by the scaling parameter: the number of cache states.
- With a low number of cache states already, a high analysis precision can be achieved.
- For some examples, the preemption delay is an order of magnitude smaller than be quantity of used cache blocks of the preempting and useful cache blocks of the preempted task (example, PrS B and C for 1 kB in Figure 13).
- There exists a cache size where the preemption delay is at maximum. For smaller, as well as larger caches, the preemption delay is smaller. Thus, the preemption delay could be the same for cache of different sizes.

- Associativity only marginally affects the precision of the preemption-delay bounds and the timing complexity of the analysis framework.

9. CONCLUSION

Cache memories can introduce unpredictable interference to task execution in real-time computing systems with preemptive scheduling. In this paper, we have proposed a scalable precision cache analysis for direct-mapped and associative-instruction caches. The experiments have shown that a high analysis precision is already attained for a low number of cache states, and a low time complexity. Thus, the approach is well suited for early design space exploration, as well as for highly accurate real-time performance verification, in which the number of cache states could be increased. Future work includes a preemption delay analysis for data caches and multilevel cache hierarchies.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. 1988. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.

AUDSLEY, N. C., BURNS, A., RICHARDSON, M. F., AND WELLINGS, A. J. 1991. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*. IEEE, Los Alamitos, CA.

BASUMALLICK, S. AND NILSEN, K. 1994. Cache issues in real-time systems. In *Workshop on Language, Compiler and Tool Support for Real-Time Systems*. ACM SIGPLAN, Orlando, FL.

BUSQUETS, J. V., SERRANO, J. J., AND WELLINGS, A. 1997. Hybrid instruction cache partitioning for preemptive real-time systems. In *Euromicro Workshop on Real-Time Systems*. IEEE Los Alamitos, CA.

BUSQUETS-MATAIX, J. V. AND WELLINGS, A. 1996. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. IEEE, Los Alamitos, CA. 204–212.

BUSQUETS-MATAIX, J. V., GIL, D., GIL, P., AND WELLINGS, A. 2000. Techniques to increase the schedulable utilization of cache-based preemptive real-time systems. *Journal of System Architecture 46*, 357–378.

CAMPOY, A., PUAUT, I., IVARS, A., AND MATAIX-BUSQUETS, J. 2005. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *Euromicro Conference on Real-Time Systems*. Palma de Mallorca, Spain.

CAMPOY, M., IVARS, A. P., AND BUSQUETS-MATAIX, J. V. 2001. Static use of locking caches in multitask preemptive real-time systems. In *IEEE Real-Time Embedded System Workshop*.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1997. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

CORTI, M., BREGA, R., AND GROSS, T. 2000. Approximation of worst-case execution time for preemptive multitasking systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*. Vancouver, Canada.

DATTA, A., CHOUDHURY, S., BASU, A., TOMIYAMA, H., AND DUTT, N. 2001. Satisfying timing constraints of preemptive real-time tasks through task layout technique. In *IEEE VLSI Design*. 97–102.

INFINEON. 2004. Tricore manual http://www.infineon.com.

JOSEPH, M. AND PANDYA, P. 1986. Finding response times in a real-time system. *The Computer Journal (British Computer Society) 29*, 390–395.

KIRK, D. B. 1989. SMART(strategic memory allocation for real-tim) cache design. In *Real-Time Systems Symposium*. IEEE Computer Society Press, Los Alamitos, CA. 229–239.

LEE, C.-G., HAHN, J., AND ET AL., Y.-M. S. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers 47,* 6 (June), 700–713.

LEE, C.-G., LEE, K., AND ET AL., J. H. 2001. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering 27,* 9 (Nov.), 805–826.

LEHOCZKY, J., SHA, L., AND DING, Y. 1989. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. 10th Real-Time Systems Symposium*. IEEE, Los Alamitos, CA. 166–171.

LIEDTKE, J., HÄRTIG, H., AND HOHMUTH, M. 1997. Os-controlled cache predictability for real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*. Montreal, Canada.

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal ACM 20,* 1 (Jan.), 46–61.

MITRA, T., NEGI, H. S., AND ROYCHOUDHURY, A. 2003. Accurate estimation of cache-related preemption delay. In *ACM/IEEE International Symposium on Hardware/Software Codesign and System Synthethis (CODES+ISSS)*. Newport Beach, CA.

MOGUL, J. C. AND BORG, A. 1991. The effect of context switches on cache performance. In *Conference on Architectural Support for Programming Languages and Operating Systems*. Santa Clara, CA. ACM, New York. 75–84.

MUELLER, F. 1995. Compiler support for softwarebased cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*. La Jolla, CA.

MUELLER, F. 2000. Timing analysis for instruction caches. *Real-Time Systems Journal 18,* 2/3 (May), 209–239.

PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 1999. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publ. Norwell, MA.

PETTERS, S. M. AND FÄRBER, G. 2001. Scheduling analysis with respect to hardware related preemption delay. In *Workshop on Real-Time Embedded Systems (Satellite Workshop of IEEE Real-Time Systems Symposium)*. London, UK.

PUAUT, I. AND DECOTIGNY, D. 2002. Low-complexity algorihtms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*.

RAMAPRASAD, H. AND MUELLER, F. 2005. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time and Embedded Technology and Applications Symposium*. 148–157.

RAMAPRASAD, H. AND MUELLER, F. 2006. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Real-Time and Embedded Technology and Applications Symposium*.

SCHNEIDER, J. 2000. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *21st IEEE Real-Time Systems Symposium*. 195–204.

SEBEK, F. 2001. Measuring cache related pre-emption delay on a multiprocessor real-time system. In *Real-Time Embedded Systems Workshop*. London.

STAPPERT, F. 2003. Wcet benchmarks. http://www.c-lab.de/home/de/people/people.php?id = Stappert_Friedhelm_00.

STASCHULAT, J. AND ERNST, R. 2004. Multiple process execution in cache related preemption delay analysis. In *International Workshop on Embedded Software (EMSOFT)*. Pisa, Italy. ACM, New York.

STASCHULAT, J. AND ERNST, R. 2006. Worst case timing analysis of input dependent data cache behavior. In *Euromicro Conference on Real-Time Systems*. Dresden, Germany.

STASCHULAT, J., SCHLIECKER, S., AND ERNST, R. 2005. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *EUROMICRO Conference on Real-Time Systems*. Palma de Mallorca, Spain.

TINDELL, K., BURNS, A., AND WELLINGS, A. 1994. An extendible approach for analysing fixed priority hard real-time systems. *Journal of Real-Time Systems 6*, 2 (Mar.), 133–152.

VERA, X., LISPER, B., AND XUE, J. 2003. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*.

Received September 2005; revised April 2006; accepted June 2006