Multiple Process Execution in Cache Related Preemption Delay Analysis

Jan Staschulat, Rolf Ernst
Technical University of Braunschweig
Institute of Computer and Communication Network Engineering
D-38106 Braunschweig, Germany
{staschulat,ernst}@ida.ing.tu-bs.de

ABSTRACT

Cache prediction for preemptive scheduling is an open issue despite its practical importance. First analysis approaches use simplified models for cache behavior or they assume simplified preemption and execution scenarios that seriously impact analysis precision. We present an analysis approach which considers multiple executions of processes and preemption scenarios for static priority periodic scheduling. The results of our experiments show that caches introduce a strong and complex timing dependency between process executions that are not appropriately captured in the simplified models.

Categories and Subject Descriptors: B.3.3: Worst-case analysis.
General Terms: Algorithms, Measurement, Performance, Verification.
Keywords: Worst Case Execution Time Analysis, Cache, Embedded Systems, Scheduling.

1. INTRODUCTION

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

There are several approaches to make caches more predictable and efficient. One approach is to partition the cache sets and to reserve these partitions for individual processes. This has been investigated in [21]. The advantage is that cache lines do not have to be reloaded after interrupts and between consecutive executions of the same process. Also, cache behavior becomes (partly) orthogonal for processes and therefore more predictable. In [6] process layout techniques are suggested which aim at minimizing the inter-process interference in the instruction cache. Another approach is to lock frequently used cache lines. Such techniques have been investigated by [14] [5] . Both approaches come at an area and power cost as they require a sufficient cache associativity to become effective. Therefore, heterogeneous memory architectures with caches and scratch-pad SRAM have been introduced [12], where the scratch-pad can hold frequently used cache lines. [19] has proposed compiler techniques for such architectures.

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

Simplified approaches extend the known RMA with fixed context switch costs [3], while more recent approaches use data flow analysis of the preempted and preempting process to bound the number of replaced cache blocks [16] [18]. However, these approaches model only a single process activation and assume an empty cache at process start thereby neglecting that cache blocks might be available for later executions. Pre-runtime scheduling heuristics which take the effects of process switching on processor cache into account have been presented in [13]. However, only non-preemptive scheduling based on the earliest deadline first strategy is considered, which is much easier than the preemptive case.

Those approaches that do take multiple preemptions into account, e.g. [18] [16] [26], bound the cache related delay for multiple preemptions pessimistically by the product of the maximum preemption cost and the number of preemptions. Our own experiments have shown that the actual cost for such a preemption scenario is much smaller than found by such approximations [24].

In this paper, we present a new analysis approach to determine the cache related preemption delay (CRPD) which considers multiple executions of processes as well as preemption scenarios for instruction caches. The approach supports hard real-time system analysis, but can also be useful for rapid design space exploration.

This paper is organized as follows. Sec. 2 describes the cache effects due to a preemption and Sec. 3 reviews related work. In Sec. 4 we describe our new refined analysis for multiple executions of processes and an integrated analysis for multiple preemptions. Experiments are presented in Sec. 5, before we conclude in Sec. 6.

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

2. PROBLEM DESCRIPTION

This work considers a single processor system with preemptive scheduling. In Figure 1 a process P_2 is activated and finishes execution without preemption. Then a lower priority process P_3 executes and P_2 is activated again. Another process P_1, which has a higher priority than P_2 preempts P_2. P_1 finishes execution, and P_2 resumes.

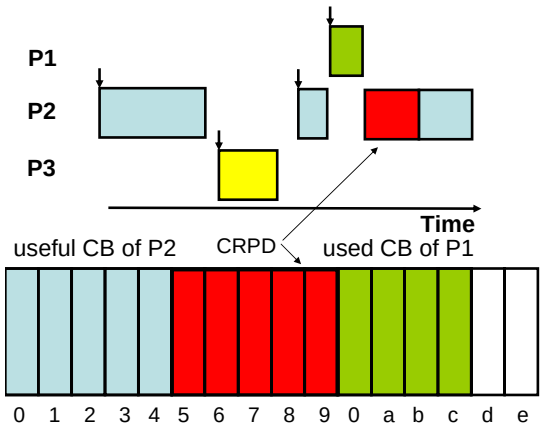

Figure 1: Scheduling of three tasks P_1, P_2 and P_3. The upper portion displays teh preemption of task P2 by P1 and the lower portion shows the cache contents.

This preemption can take place anytime during execution of P_2. In rate monotonic scheduling shorter processes have a higher priority which lead to multiple preemptions of a lower priority process, such as P_2. In the lower part of Fig. 1, a direct mapped cache with 16 cache blocks is shown. The cache behavior during the preemption is described in Sec. 2.3. In this paper we only analyze instruction caches.

2.1 Application domain

Current analysis techniques, which are reviewed in Sec. 3, model each process in isolation and assume pessimistically an empty cache at process start. The WCET of the process is overestimated, because compulsory cache misses might not be necessary for the next activation.

This might possibly be acceptable for processes with computation intensive loops such as those found in large filter algorithms, MPEG decoding or sorting algorithms. However, typical automotive real time applications, such as engine control, consist of linear code without loops that are activated periodically by the operating system. This property is not limited to automotive applications, as those generated from Matlab/Simulink, Petri-nets or ASCET/SD often posses this property.

This linearity leads to a fundamentally different cache behavior. A cache does not speed up linear code, since memory lines are executed sequentially and only once. The speedup is gained only if the cache holds memory lines from an earlier process activation. In the best case, all memory lines are present in the cache and a second activation requires no further instruction cache misses.

The second drawback of current approaches is that only higher priority processes are considered in CRPD- and scheduling analysis. But this simplification is only acceptable if an empty cache is assumed at every process start. It is not acceptable for processes with linear code which are activated multiple times: the worst case response time will be overestimated and unsuitable for the design and verification of realistic embedded systems. Therefore we developed a new approach, which considers the cache state at process start.

Consider Fig. 1. After the first activation of P_2, some cache blocks can be replaced by the lower priority process P_3 and available cache blocks for the second activation of P_2 reduce the number of compulsory cache misses.

Preemptive scheduling analysis has to consider the WCET, direct context switch costs of the operation system and the indirect cache related preemption delay. The CRPD depends on the frequency and location of preemptions as well as the additional time delay for reloading replaced cache blocks.

2.2 Preemption frequency and location

The number of preemptions depends essentially on the scheduling strategy and the operating system. In this paper we consider only static priority periodic scheduling and assume the number of preemptions to be known a priori. This is not a limitation, since the approach can be coupled with an iterative response time analysis, as proposed for a wide range of scheduling strategies, for example [4].

Preemptions can take place anytime. Therefore a lower priority process can be preempted anywhere except where preemptions are explicitly disabled, such as in protected program segments. The space of possible preemption points is given by the control flow graph (CFG) of the process, where nodes represent basic blocks of the process and edges the control flow dependency. In this work we assume at most one preemption at each basic block, as in [15]. We argue in Sec. 4.1 why this is correct, even though several preemptions might take place in long basic blocks.

2.3 Preemption cost

The time delay for one preemption depends on the preempted process P_2, the preempting process P_1, and the instruction cache state at the start of process P_2.

Only *useful* cache blocks can lead to additional cache misses, where a useful cache block at an execution point is defined as a cache block that contains a memory block that *may* be re-referenced before being replaced by another memory block [16]. For example, it is possible that the replaced memory block is one that is no longer needed or one that will be replaced without being re-referenced, even when there were no preemptions. The number of useful cache blocks depends on the control flow structure. All cache blocks that hold memory blocks of a loop body are useful, provided that the entire loop body fits in the cache. The number of useful cache blocks within a basic block of a sequential process is at most one at every execution point if an empty cache is assumed at process start. Additionally, one cache miss occurs if a preemption replaces the cache block, which contains the instructions of the current basic block.

The second influence is induced by the preempting process. Only the cached memory blocks of process P_1 replace cache blocks of process P_2. The execution path of P_1 that uses the maximum number of cache blocks has been considered as the worst case for the preempting process [26].

The worst case CRPD for one preemption is given by the intersection of the maximum number of useful cache blocks of process P_2 and the used cache blocks of process P_1 multiplied by the constant cache refill time at a given preemption point p_1. In our example the preemption cost $CRPD_{P_2}^{P_1}(p_1)$ where process P_1 preempts P_2 at preemption point p_1 is defined by

$$CRPD_{P_2}^{P_1}(p_1) = t_{refill} \cdot |UCB_{P_2}(p_1) \cap UB_{P_1}| \qquad (1)$$

where t_{refill} denotes the cache refill time, $UCB_{P_2}(p_1)$ the set of useful cache blocks of process P_2 at p_1 and UB_{P_1} the set of used cache blocks of process P_1.

Fig. 1 presents the cache contents for P_1 and P_2. Cache blocks 0 -9 are useful cache blocks (CB) of P_2 at the preemption point and CB 5 - c are the used cache blocks of P_1. The intersection (CB 5-9) are the cache blocks that are reloaded when P_2 resumes execution. In this case the CRPD is five cache misses. Then, for a given number n of preemptions, the total CRPD assumed by [16] [18] [26] is given by

$$n \cdot max_{p_i} CRPD_{P_2}^{P_1}(p_i) \qquad (2)$$

The drawback of such a pessimistic model is using the maximum time delay for every preemption. This greatly overestimates the actual preemption cost. We have seen in experiments that the preemption cost tends to drop significantly for multiple preemptions [24]. Consider two preemption points p_1 and p_2 with the same number of useful cache blocks (not shown in the figures). If a preemption at p_1 replaces them all then a preemption at p_2 cannot replace even one, because all useful cache blocks have already been replaced. So the preemption cost at p_2 is zero. In general, the preemption cost of the nth preemption depends on the replaced cache blocks of all $n-1$ previous preemptions. This will be further analyzed in Sec. 4.3. Before we present our approach in Sec. 4 we review related work.

3. RELATED WORK

Early work on cache behavior for a single process does not take preemption into account [17] [9] [28] [27]. First proposals for cache modeling and timing analysis use simplified cache models. [2], [3] and [20] extend the known RMA by a fixed context switch cost. [15] uses data flow analysis to determine the CRPD when a process P_1 preempts process P_2 by analyzing the number of useful cache blocks of P_2. Then, a complex analysis follows, analyzing all possible combinations of preemptions. The cost of multiple preemptions is determined by the sum of the n most expensive preemptions assuming all useful cache blocks to be replaced. They refine their approach in [16], by intersecting the number of useful cache blocks of P_2 with the number of used cache blocks of P_1. However, to cope with the computational effort of their complex preemption model the computation of multiple preemptions is simplified to multiplying the maximum preemption cost by the number of preemptions. The number of preemptions is determined by integer linear programming (ILP) and process phasing based on worst case and best case response time (BCRT) of processes. However, the BCRT analysis is a complicated problem where only approximative solutions have been proposed for the general case ([11] [10]) and the BCRT determination is not described by the authors. Unfortunately, they don't publish experiments showing the accuracy of this model. Furthermore, both versions assume an empty cache at process start and analyze each preemption separately. Thus, the important case of multiple process executions is not considered.

Another approach [26] concentrates on the analysis of used cache blocks of preempting process P_1 using an ILP technique and classifies all cache blocks of the preempted process P_2 as useful. Multiple preemptions are not considered and a cold cache is assumed at process start. [18] refines the data flow analysis of [16] and extend the approach of [26] by modeling the cache content as a *state* instead of a set. All possible cache states of the preempting and preempted process are intersected to find the maximum CRPD. Multiple preemptions are not considered and an empty cache is assumed at process start.

[22] uses the same worst case assumption based on cache analysis by [9] but uses a more precise context switch model, including deep pipeline scheduling. A simulation based method is suggested in [7], which uses live cache frames to bound the number of replaced cache blocks. Again, an empty cache at process start and a single delay for all preemptions is considered.

Currently the preempting and preempted process are accurately analyzed by data flow analysis enhanced by considering several paths in the control flow graph. However, above approaches assume an empty cache at process start. This is very pessimistic, since processes in real time embedded systems are activated multiple times. Existing cache blocks from earlier executions can reduce the number of cache misses in later executions substantially. Current approaches either model only the number and cost of preemptions or multiple process execution without preemption delay, but no approach models both situations. Only the combination of both effects provides sufficient accuracy, as we will see in the experimental results.

4. REFINED APPROACH

The refined approach addresses two aspects in cache related preemption delay analysis: (1) the cost of preemption scenarios and (2) the cache state at process start, to consider multiple process activations. A preemption scenario $(P_2, P_1, \{p_1, \cdots, p_n\})$ is a 3-tuple of the preempted process P_2, the preempting process P_1 and the set of preemption points $\{p_1, \cdots p_n\}$. A preemption point p_i is a basic block in the CFG of the process P_1, like in [16].

The assumptions of our approach are summarized in Sec. 4.1. The analysis of a single preemption is described in Sec. 4.2 for direct mapped and m-way set associative caches. Sec. 4.3 describes our approach for preemption scenarios and Sec. 4.4 for multiple process activations. Finally the computation of the worst case preemption scenario is presented in Sec. 4.5.

4.1 Assumptions

Our approach has five main assumptions:

1. The worst case scenario does not contain two preemptions in the same execution of a basic block. This simplification can be justified by the following consideration. Suppose thee are m preemptions during the execution of a large basic block b_i. The total preemption cost is then bounded by

$$C = maxCRPD(b_i) + m \qquad (3)$$

cache misses. That is, the maximum cache related preemption delay of b_i plus m cache misses for the m

preemptions of the block that possibly require reload of the current cache block when that basic block is continued. The reason is that a basic block consists only of sequential code and at most one cache block is useful at any time during its execution. Therefore, the maximum CRPD for each additional basic block preemption is equal to or less then any other preemption in the process. As long as there are still basic blocks that have not been preempted, this assumption does not change the maximum preemption cost. This assumes that there are more basic blocks than preemptions by another process, which we consider to be true for real-life control flow graphs. It is obviously easy to detect situations in which this assumption does not hold and equally easy to add the corresponding 1 cache miss per additional preemption. So, this assumption is not a limitation.

2. The number of preemptions is given a priori, but this number can formally be bounded by the response time analysis for scheduling algorithms other then fixed priority periodic scheduling, e.g. [4] [3].

3. All processes of the system run between two process activations. This is the worst case for multiple process activations. Future research is necessary for a more precise bound.

4. The system uses an m-way associative instruction cache with a deterministic refill strategy (LRU, FIFO) but not a random strategy. In this paper we only analyze the instruction cache.
5. A constant delay time t_{refill} is assumed for a cache miss.

4.2 Single preemption cost

To calculate the CRPD for a process, we intersect the set of useful cache blocks of the preempted process with the set of used cache blocks of the preempting process by extending the cache state approach of [18].

4.2.1 Direct mapped caches

This subsection describes the approach of [18] for direct mapped caches. A cache state denotes the contents of all cache blocks. For a direct mapped cache with n blocks, a cache state is a vector of n elements, where $c[i] = m$ if cache block i contains memory block m. A reaching cache state RCS_B at a basic block B of a process is the set of possible cache states when B is reached via any incoming program path. The live cache states at a basic block B, denoted LCS_B, are the possible first memory references to cache blocks via any outgoing program path from B. A least fixed point algorithm computes the values of these sets. To compute RCS, the quantities RCS_B^{IN} and RCS_B^{OUT} are computed and we set $RCS_B = RCS_B^{OUT}$ if the fixed point is reached. Initially $RCS_B^{IN} = \emptyset$ and $RCS_B^{OUT} = gen_B$, where gen_B holds all memory blocks introduced into the cache by basic block B. The iterative equations are as follows:

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

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

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

Similarly LCS is computed by an iterative fixed point algorithm. RCS_B captures the possible cache states when P is preempted and LCS_B captures the possible cache usages when P resumes execution. The intersection of both sets is the set of useful cache blocks of basic block B. The used cache blocks of a preempting process P' is given by RCS_{end}, assuming *end* is the last basic block of P'. Finally the CRPD at a basic block B is computed by the intersection of used cache blocks and useful cache blocks.

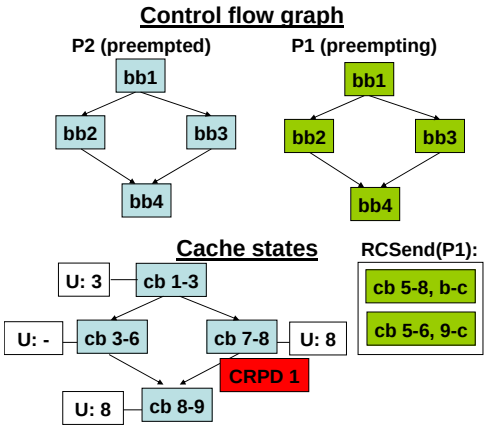

Figure 2: Analysis of useful and used cache blocks

Fig. 2 shows a small control flow graph of process P_1 and P_2, where memory lines for basic block 3 map to cache lines 7 - 8 and memory lines from basic block 4 map to cache lines 8 - b. Therefore at b_3 cache line 8 is useful. We assume that cache line 8 is also useful at b_4. On the right side the used cache blocks of process P_1 are shown. In this case two RCS_{end} states are possible at the last control flow node of P_1. Because cache line 8 is used and useful at node b_3 one cache miss would occur, if a preemption takes place at node b_3.

4.2.2 Extention for n-way associative caches

A n-way set associative caches contains sets with n cache blocks each. The RCS and LCS are defined for each cache set. The usefulness of a cache block is determined by comparing the contents of both RCS and LCS. Unlike a direct-mapped cache, in a n-way set associative cache a memory line can be placed in n different positions in a set. Hence only the cache blocks within a cache set are compared. For example, the content of the first (second, third, ...) block in set 1 in RCS is compared with the contents of all n blocks in set 1 in LCS and so on.

4.3 Multiple preemption cost

We extend the path based CRPD analysis of [18] for preemption scenarios. Suppose that $S = \{P_2, P_1, \{b_3, b_4\}\}$ denotes the preemption scenario at basic block b_2 and b_4 where P_1 preempts P_2. At process start we assume an empty cache for now. Fig. 3 shows this preemption scenario.

The cost of the first preemption is calculated by the least fixed point algorithm as described in Sec. 4.2. For all further preemptions we procede as follows: To capture the effect of

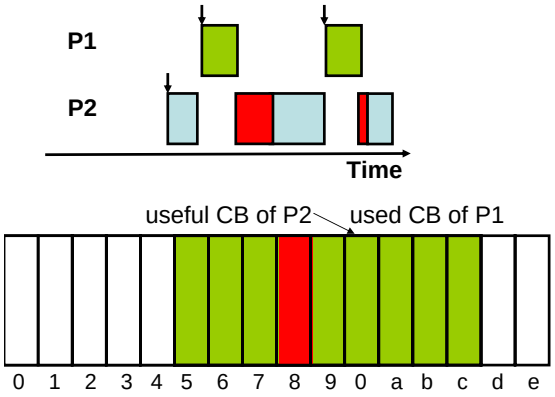

Figure 3: Second preemption of process P2 by P1

a preemption at basic block b_i by P_j on later preemptions, a data flow analysis is performed for the preempting process.

The cache states of basic block b_i are inserted at the first CFG node of P_j. Then the data flow analysis determines all reaching cache states of P_j. The cache states of the last CFG node of the preempting process P_j, RCS_{end}, is the set of used cache blocks of P_j.

For each cache state CS_k of RCS_{end}, we insert after b_i a new basic block node in the CFG of the preempted process and also insert the cache state CS_k. If the preempting process P_j finishes with n different reaching cache states then n nodes are inserted. For the inserted node N_k, we define $gen_{N_k} = RCS_k$, where RCS_k is the kth reaching cache state of the last basic block of P_j.

For our example in Fig. 1, we see in Fig 3 that only the cache block (CB) of P_2 is useful and CB 5 - c are used by P_1. Figure 4 shows a preemption at basic block b_3 which uses CB 7 and 8. The preempting process has two cache states at its last node, hence two nodes are inserted. The resulting CRPD is one, because only CB 8 is useful.

Then the iterative data flow analysis is applied and the RCS of all other nodes are calculated again. This models the fact that useful cache blocks might be overwritten by a preemption and thus cannot be replaced again. However, the LCS property is not recalculated, because otherwise it would be possible to consider the preempting process' cache blocks as useful cache blocks. After recalculating the RCSs, the CRPD is calculated for the next preemption point of the preemption scenario. This procedure is applied for every preemption point of the preemption scenario.

For loops this analysis is very complex, because several iterations have to be modeled. We can simplify the analysis by considering the number of iterations L and the total number of preemptions n . For the empty cache at process start, the maximum preemption cost will occur within a loop. If $L \geq n$ then we can precicely calculate the maximum cost by $n \cdot CRPD_{loop}$, because all replaced cache blocks are reloaded in the next iteration. $CRPD_{loop}$ denotes the maximum preemption cost within the loop body. If $L \leq n$ we can at least conservatively approximate it with $n \cdot CRPD_{loop}$. A more accurate analysis would have to unroll the loop L times and consider all possible preemption scenarios, thus increasing the number of possible combinations.

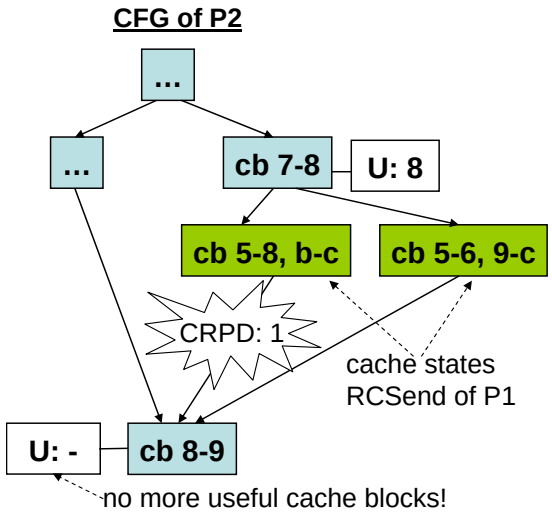

Figure 4: Insertion of nodes in CFG of process P1

4.4 Multiple process execution

In the previous sections we assumed an empty cache at process start. However, some cache blocks might be present in cache when the process is activated a second time. This effects the core execution time of the process itself as well as the CRPD.

At first we assume that no processes run between two activations of process P_i. We model a cache state like in Sec. 4.2 by inserting new nodes, but now we insert before the first node in the control flow graph k nodes for k different RCS_{end}^{OUT} values of process P_i.

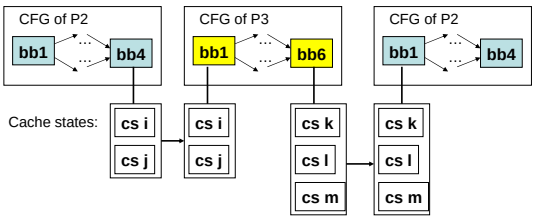

Figure 5: Multiple process activation

It is important to consider the processes which run between two activations. In this paper we assume the conservative approximation that all higher priority and lower priority processes of the system can execute. This is new to CRPD analysis, as current techniques consider only higher priority processes.

We model the cache behavior of these intermediate execution of $P_i, P_{j_1}, \cdots, P_{j_k}, P_i$ as a sequence of process executions. The sets of RCS_{end}^{OUT} of the preceding process P_{j_m} are inserted as start nodes of process $P_{j_{m+1}}$. The last process is the next instance of P_i. Note, that the CRPD at the second activation of process P_j is independent of the order and the

frequency of intermediate processes. (See [25]).

Fig. 5 shows two activations of P_2 and one intermediate lower priority process P_3 corresponding to Fig. 1. The two cache states cs_i, cs_j at the end of process P_2 are propagated to the first node of the P_3s CFG node. Then the usual data flow analysis determines the RCS states for all nodes. The cache states at the last node cs_k, cs_l, cs_m are again propagated to the beginning of P_2.

4.5 Worst case preemption scenario

This section describes how we find the preemption scenario with the maximum cache related preemption delay. We assume that the preempted process P_2, the preempting process P_1 and the number of preemptions n by P_1 during execution of P_2 is given. The analysis is based on the control flow graph which is constructed by SymTA/P for every process.

SymTA/P is a tool to determine the WCET of processes by analyzing feasible pathes on the source code level and is currently being developed at our institute. The architecture can be modeled by an off-the-shelf cycle accurate processor simulator or by measurments on an evaluation board. Further on, a cache analysis determines the worst case cache behavior for m-way associative instruction caches. The longest execution path is found by solving a linear optimization problem based on the control flow graph of the process. For details refer to [27].

Because the total preemption cost for n preemption nodes is not additive it has to be computed as described in Sec. 4.3. However, the sum of each preemption cost serves as an upper bound for the total preemption cost.

4.5.1 Branch and bound algorithm

Given a control flow graph with m nodes there are $\binom{n}{m}$ possible combinations of preemption points for n preemptions, assuming one preemption at each node. The maximum preemption cost problem is solved as a global optimization problem. We use Branch and Bound as a very flexible and efficient algorithm.

We define $C(n_i)$ as the cache related preemption cost at node i, which represents a basic block. This cost $C(n_i)$ is calculated as described in Sec. 4.2. The calculation for the cache related preemtion delay of node n_{i+1}, where preemptions that occur at the nodes $n_1 \cdots n_i$, are denoted by $C_{n_1,\cdots,n_i}(n_{i+1})$. The calculation of this term is described in Sec. 4.3.

The branch and bound algorithm starts with the computation of $C(n_i)$ for all nodes in the control flow graph. The nodes are sorted by the cost $C(n_i)$ descendently. We index the list using i, e.g. $L[i]$, is the i'th most expensive preemption node.

Next, a tree is constructed, where nodes are the preemption nodes n_i and the depth represents the number of preemptions. Nodes on the same level k are possible candidates for the kth preemption. A path from the start-node to a leaf node represents a preemption scenario. To initialize the tree, all nodes n_i of the CFG, ordered by the cost $C(n_i)$, are inserted at the same level after the empty start node.

Assuming that the total cost for i preemptions $\tilde{C}(n_1, \cdots, n_i)$ is given, the total cost $\tilde{C}(n_1, \cdots, n_i, n_{i+1})$ for $i+1$ preemptions is computed by

$$\tilde{C}(n_1, \cdots, n_i, n_{i+1}) = \tilde{C}(n_1, \cdots, n_i) + C_{n_1,\cdots,n_i}(n_{i+1}) \quad (7)$$

E.g., the total cost of all i previous preemptions $\tilde{C}(n_1, \cdots, n_i)$ plus the preemption cost for node n_{i+1} assuming preemptions at the nodes $n_1, \cdots, n_i$.

The quality of this algorithm depends on the branch-and bound conditions. Branching is controlled by the single preemption cost $C(n_i)$. For a new candidate a node is chosen, which is not taken, e.g., not on this path on the higher levels of the search tree, and has maximum cost, e.g. $\{n_k | C(n_k) maximum, n_k \in L\}$. The initial bound B of the CRPD for n preemptions is given by iterating equation 7 exactly n times and choosing the most expensive node that is not yet taken.

A sub-tree is bounded if the estimated cost after $n+1$ preemption points, C^{i+1}_{total} of eq. 8, is smaller than the current bound B. This means no preemption node is inserted and the search continues at the $i-1$th level.

$$C^{i+1}_{total} = \tilde{C}(n_1, \cdots, n_i, n_{i+1}) + \sum_{k=1}^{n-i-1} C(\{L \setminus \{n_1 \cdots n_{i+1}\}[k]) \quad (8)$$

The cost C^{i+1}_{total} denotes the sum of the cost of previous preemptions including the current one, $\tilde{C}(n_1, \cdots, n_i, n_{i+1})$, and the upper bound of the cost of subsequent preemptions, e.g., the sum of the most expensive $n-i-1$ preemption nodes which are not yet taken. If $C_{total} \geq B$ then the node n_{i+1} is inserted at the node n_i on the $i+1$th level and the search continues until n preemption points are chosen. At the nth node, the leaf node, the current bound B is replaced by C^n_{total}, if it is larger than B.

4.5.2 Multiple process activations

For multiple process activation at first the reaching cache states of the preempted process and the intermediate processes is analyzed as described in Sec. 4.4. Then the worst case preemption scenario for n preemptions is computed as described in Sec. 4.5.1.

4.6 Algorithmic complexity

This section summarizes the complexities of the proposed algorithms. The data flow analysis of Sec. 4.3 to compute the number of useful cache blocks for multiple preemptions may increase exponentially with the number of inserted cache states RCS_{out} of the preempting process. The analysis for the preempting process is also cache state based, so there are as many RCS states as paths with different cache behavior. This is a critical issue of our approach and we are currently developing approximation heuristics.

For multiple process activations, the complexity does not grow because only a fixed number of cache states are inserted at the start node.

The estimation of the most expensive preemption scenario is theoretically exponential, but our experimental results of the branch and bound algorithm of Sec. 4.5.1 indicate that the number of paths to be investigated can be bounded very effectively.

Table 1 presents the number of preemption scenarios for 1 to 4 preemptions for two benchmarks, FFT with 28 basic blocks and FFT with 99 basic blocks. For 4 preemptions the number of preemptions considered by branch and bound compared with a all combinations is 1.65% for FIR and 0.61% for FFT, which is promising for larger benchmarks.

preemptions	FIR		FFT	
n	B&B	$\binom{n}{28}$	B&B	$\binom{n}{99}$
1	28	28	99	99
2	175	378	390	4851
3	222	3276	9516	156849
4	337	20475	23014	3764376

Table 1: Number of preemption scenarios for branch and bound algorithm compared with total number of combinations

5. EXPERIMENTS

In this section we present the accuracy and performance of our CRPD analysis technique for multiple preemptions and multiple activations.

5.1 Experimental setup

Name	Mem[B]	Description
sqrt	94	square root calculation [18]
dac	168	array calculation with loops [27]
linear1	172	sequence of 10 add instructions
linear2	652	sequence of 40 add instructions
nsich	804	car window lift control [23]
statemate	872	car window lift control [23]

Table 3: Benchmark name, memory size in Byte and description

We select six different benchmarks for our experiments (refer to Table 3). We use the ARM developer studio[1] for processor simulation and Dinero[8] for cache simulation. All benchmarks are compiled for ARM946 assembly language with fixed four byte instruction width. The control flow graph is generated from C code with SymTA/P. Given the CFG and the ARM memory mapping file, our analyzer computes the CRPD. The cache parameters, preempted process, preempting process, number of preemptions and the preemption scenario are defined by an XML description. `sqrt` and `dac` are the only C programs with loops. `nsich` and `statemate` were generated by STAtechart Real-time-Code generator STARC, C-lab [23] which specifies an car window lift control.

5.2 Multiple process activation

First we show the accuracy of our modeling for multiple process activation. We choose `dac`, `sqrt`, `linear2`, and `statemate` as higher priority tasks and `linear` and `nsich` as higher priority tasks. Table 2 presents the total number of cache misses during the second activation of the preempted task for different cache configurations. The first column shows the cache parameters. A 2-way associative 256 Byte cache with block size 8, for example, is denoted as 512-8-2. For each preemption pair P_1/P_2 the results from the conservative approximation C_{negi} by [18], from our refined analysis C_{ana}, and from exhaustive simulation C_{sim} is given. For the first two pairs the preempted process contains loops, for last two columns the preempted process contains only linear code. C_{negi} is calculated by

$$C_{negi} = CM_{P_1} + CRPD_{P_1}^{P_2} \quad (9)$$

where CM_{P_1} denotes the total number of cache misses of task P_1 during execution starting with an empty cache, which is estimated by simulation. $CRPD_{P_1}^{P_2}$ denotes the cache related preemption delay. It is calculated by our refined analysis with the empty cache at process start configuration. C_{ana} is calculated by

$$C_{ana} = CM_{P_1P_1} + CRPD_{P_1P_1}^{P_2} \quad (10)$$

where $CM_{P_1P_1}$ denotes the number of cache misses during the second activation of P_1, supposing that P_1 was executed once before and is estimated by simulation. $CRPD_{P_1P_1}^{P_2}$ denotes the cache related preemption delay for the second activation of P_1 by P_2 as the result of our approach in Section 4.4. For simplicity we assume that no process runs between two process activations of P_1.

The results show that our approach is very close to the actual number of cache misses determined by simulation. The conservative approximation by [18] is for some cache architectures 100% inaccurate. This is because an empty cache is assumed for every process activation. The conservative approach is also highly innacurate for larger caches, where all applications fit entirely in the cache. For linear programs this is the case for 2KB caches and for programs with loops it is the case for the 2-way 1KB and 2-way 2KB cache. The performance of our analysis for the benchmarks was between 30 seconds and 8 minutes on3.4 GHz Pentium 4 processor and 2 GB RAM.

Let us now consider the effect of of multiple preemptions regarding the response time of the preempted process. Table 4 presents the response time for different cache architectures and benchmarks in terms of clock cycles (clk). With the ARM simulator we determine the core execution time t_{P_1}, t_{P_2} of process P_1 and P_2 respectively. t_{resp}^{negi} and t_{resp}^{ana} are given by equation 12 and 13.

$$X = t_{core}^{P_1} + t_{core}^{P_2} + I_1 + I_2 + CM_{P_2}(P_m - 1) \quad (11)$$

$$t_{resp}^{negi} = X + C_{negi}(P_m - 1) \quad (12)$$

$$t_{resp}^{ana} = X + C_{ana}(P_m - 1) \quad (13)$$

where I_1 and I_2 are the number of executed instructions of P_1 and P_2 and P_m the cache miss penalty (not shown in Table 4). C_{negi} and C_{ana} are defined by equation 9 and 10. The response time is calculated by adding the core execution times and the time for cache hits and misses for preempted process P_1 and cache hits and misses for preempting process P_2. The term $I_1 + C_{negi}(P_m - 1)$ for t_{resp}^{negi} is the delay of cache misses ($C_{negi} \cdot P_m$) and cache hits ($I_1 - C_{negi} \cdot 1$). For our experiments we assumed one clock cycle for a cache hit and $P_m = 20$ clock cycles for a cache miss.

The results show that the response time is pessimistically overestimated by Negi's approach. The last column presents the performance loss $P_{loss}^{negi} = \frac{t_{resp}^{negi} - t_{resp}^{sim}}{t_{resp}^{sim}}$, which could be gained with a more accurate analysis. The inaccuracy grows with the cache size. For example, the performance loss in case of a 1KB and 2KB cache for `sqrt/linear` is 70% and for `statemate/nsich` even 83% for the 2KB cache. The results for our refined analysis is in most cases exact to the simulated response time, the maximum error is 4% in the case of `sqrt/linear` for direct mapped 512 instruction cache.

5.3 Multiple preemption cost

Now we consider multiple preemptions with an empty and preloaded cache. Table 5 presents the preemption cost of

Cache-C.	dac/linear			sqrt/linear			linear2/nsich			statem./nsich		
	C_{negi}	C_{ana}	C_{sim}	C_{negi}	C_{ana}	C_{sim}	C_{negi}	C_{ana}	C_{sim}	C_{negi}	C_{ana}	C_{sim}
256-8-1	20	12	12	60	38	30	83	83	83	100	100	100
256-8-2	16	6	6	60	48	42	83	83	83	100	100	100
512-8-1	20	12	12	52	22	19	83	83	83	100	100	100
512-8-2	12	0	0	52	22	19	83	83	83	100	100	100
512-16-1	13	9	8	28	11	10	43	44	43	51	53	51
512-16-2	8	0	0	22	9	9	43	44	43	51	53	51
1024-8-1	12	12	12	52	22	19	83	69	67	100	63	60
1024-8-2	12	0	0	42	0	0	83	70	68	100	85	84
2048-8-1	12	12	12	52	22	19	83	0	0	100	0	0
2048-8-2	12	0	0	42	0	0	83	0	0	100	0	0

Table 2: Number of total cache misses for one preemption at second process activation.

Benchmark	C-size	$t_{core}^{P_1}$ [clk]	$t_{core}^{P_2}$ [clk]	t_{resp}^{negi} [clk]	t_{resp}^{ana}[clk]	t_{resp}^{sim} [clk]	P_{loss}^{negi}[%]
dac/linear	256-8-1	197	42	1193	1041	1041	15
dac/linear	512-8-1	197	42	1193	1041	1041	15
dac/linear	1024-8-2	197	42	1041	813	813	28
sqrt/linear	512-8-1	384	42	2119	1549	1492	42
sqrt/linear	1024-8-2	384	42	1929	1131	1131	70
sqrt/linear	2048-8-2	384	42	1929	1131	1131	70
linear2/nsich	1024-8-1	163	286	3336	3070	3032	10
linear2/nsich	2048-8-1	163	286	4269	2690	2690	58
linear2/nsich	2048-8-2	163	286	4269	2690	2690	58
statemate/nsich	512-8-1	241	286	4174	4174	4174	0
statemate/nsich	1024-8-1	241	286	4174	3585	3547	18
statemate/nsich	2048-8-1	241	286	4174	2274	2274	83

Table 4: Response time for a preemption during second activation for several benchmarks and cache sizes

five preemptions for four task sets with `dac`, `sqrt`, `nsich`, and `statemate` as lower priority tasks and for simplicity we choose the linear benchmarks `linear` and `linear2` as higher priority tasks. The results show that for an empty cache our

LP / HP Task	Cache	Empty Cache		Preloaded Cache	
		Negi	Ana	Negi	Ana
dac/linear	512-8-1	52	52	52	43
dac/linear	1024-8-1	52	52	52	40
dac/linear	1048-8-1	12	12	12	12
sqrt/linear	512-8-1	182	182	182	169
sqrt/linear	1024-8-1	47	47	47	1
sqrt/linear	2048-8-1	42	42	42	0
nsich/linear2	512-8-1	104	104	104	83
nsich/linear2	1024-8-1	104	104	104	74
nsich/linear2	2048-8-1	99	99	99	0
statemate/linear2	512-8-1	125	125	125	107
statemate/linear2	1024-8-1	125	125	125	97
statemate/linear2	2048-8-2	120	120	120	0

Table 5: Comparison of total number of cache misses of Negi and our approach for 5 preemptions with empty and preloaded cache for given lower priority (LP) and higher priority (HP) tasks.

analysis does not improve the accuracy for processes with or without loops. The reason is that the most expensive preemption points are inside the loop body. In the above benchmarks the number of loop iterations was greater than five, therefore a all preemptions occurred in the loop body.

However, in the case of multiple activations our analysis approach yield more accurate results because the effect of a preemption is propagated in the control flow graph. For a larger 2 KB cache the preemption cost is even zero for all preemptions in benchmark `nsich/linear2` and `statemate /linear2`, in contrast to 99 and 120 cache misses in Negi's approach.

The performance of our analysis ranged from several minutes for `dac/linear` and `sqrt/linear` to several hours for `nsich/linear2` and `statemate/linear2`. The reason for the long running time is the exponential number of states that are propagated after inserting a preemption node.

6. CONCLUSION

In this paper we have proposed a refined cache related preemption delay analysis which considers multiple process activations and preemption scenarios. The proposed technique extends the approach of [18] by propagating replaced cache blocks in the control flow graph and extending the data flow analysis for m-way associative instruction caches. Multiple process activations are modeled by inserting an edge from the last to the first node.

The results with a realistic processor architecture show that cache effects lead to process interdependencies which can easily outweigh individual process execution times. Such cases are not covered by the classical performance analysis approaches which are based on individual process execution times plus independent blocking times (e.g. [4]).

Further research is necessary to develop less complex analysis algorithms for multiple preemptions, to analyze the set of processes that execute between two process activations, to consider data caches and to integrate the CRPD analysis with the response time analysis.

Also, applications with loops behave better in other approaches. However, for automotive control applications linear code is very important (e.g. Matlab/Simulink generated

code). Here current approaches result in large overestimations. On the other hand, cache parameters have a significant influence on process interdependence. We can therefore conclude that cache analysis should receive maximum attention in embedded system design, process systems should be used as benchmarks rather than individual processes to consider multiple process activation and that new models and approaches are needed for performance analysis of systems with caches.

7. REFERENCES

[1] ARM Developer Suite, (ADS) version 1.2. *http://www.arm.com/devtools.ns4/html/ADS*.

[2] Swagato Basumallick and Kelvin Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

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

[4] G. Buttazzo. *Hard Real-Time Computing Systems*. Norwell, MA: Kluwer, 1997.

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

[6] Anupam Datta, Sidharth Choudhury, Anupam Basu, Hiroyuki Tomiyama, and Nikil Dutt. Satisfying timing constraints of preemptive real-time tasks through task layout technique. In *Proceedings of 14th IEEE VLSI Design*, pages 97–102, January 2001.

[7] Harry Dwyer and John Fernando. Establishing a tight bound on task interference in embedded system instruction caches. In *CASES'01*, Atlanta, Georgia, USA, November 16-17 2001.

[8] Jan Edler and Mark D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/ markhill/DineroIV.

[9] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 1999.

[10] J. C. Palencia Gutierrez, J. J. Gutierrez Garcia, and M. Gonzalez Harbour. Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems. In *Proceedings of 10th Euromicro Workshop on Real-Time Systems*, pages 35–44. IEEE Computer Society Press, June 1998.

[11] William Henderson, David Kendall, and Adrian Robson. Improving the accuracy of scheduling analysis applied to distributed systems computing minimal response times and reducing jitter. *Real-Time Systems*, 20(1):5–25, 2001.

[12] Infineon. Tricore 1 manual http://www.infineon.com/cgi /ecrm.dll/ecrm/scripts/prod_ov.jsp?oid=30926&cat_oid=-8362.

[13] Daniel Kästner and Stephan Thesing. Cache sensitive pre-runtime scheduling. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 1998.

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

[15] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on computers*, 47(6):700–713, June 1998.

[16] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on software engineering*, 27(9):805–826, November 2001.

[17] Sharad Malik and Yau-Tsun Steven Li. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[18] Hemendra Sigh Negi, Tulika Mitra, and Abhaik Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS'03*, Newport Beach, California, USA, October 1-3 2003.

[19] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers,Norwell, MA, 1999.

[20] Stefan M. Petters and Georg Färber. Scheduling analysis with respect to hardware related preemption delay. In *In Workshop on Real-Time Embedded Systems*, London, United Kingdom, December 3 2001. (Satellite Workshop of The IEEE Real-Time Systems Symposium (RTSS 2001)).

[21] Isabelle Puaut and David Decotigny. Low-complexity algorihtms ofr static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002.

[22] Jörn Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *21st IEEE Real-Time Systems Symposium*, pages 195–204, November 2000.

[23] Friedhelm Stappert. Wcet benchmarks. http://www.c-lab.de/home/de/people/people.php?id=Stappert_Friedhelm_00.

[24] Jan Staschulat and Rolf Ernst. Cache effects in multi process real-time systems with preemptive scheduling. Technical report, IDA, TU Braunschweig, Germany, November 2003.

[25] Jan Staschulat and Rolf Ernst. Crpd independence for multiple process execution. Technical report, IDA, TU Braunschweig, March 2004.

[26] Hiroyuki Tomiyama and Nikil D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *ACM International Symposium on Hardware Software Codesign (CODES)*, 2000.

[27] Fabian Wolf. *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers, 2002.

[28] Fabian Wolf, Jan Staschulat, and Rolf Ernst. Hybrid cache analysis in running time verification of embedded software. *Design Automation for Embedded Systems*, 2002.