WCET ANALYSIS FOR PREEMPTIVE SCHEDULING[1]

Sebastian Altmeyer[2], Gernot Gebhard[3]

Abstract
Hard real-time systems induce strict constraints on the timing of the task set. Validation of these timing constraints is thus a major challenge during the design of such a system. Whereas the derivation of timing guarantees must already be considered complex if tasks are running to completion, it gets even more complex if tasks are scheduled preemptively – especially due to caches, deployed to improve the average performance. In this paper we propose a new method to compute valid upper bounds on a task's worst case execution time (WCET). Our method approximates an optimal memory layout such that the set of possibly evicted cache-entries during preemption is minimized. This set then delivers information to bound the execution time of tasks under preemption in an adopted WCET analysis.

1. Introduction

Validation of hard real-time systems strongly relies on safe estimations of upper bounds on a task's worst case execution time (WCET). Computing such a WCET bound is already a complex problem for non-preemptively scheduled tasks. It becomes even more problematic in a preemptive environment. This means that the flexibility of a preemptive schedule comes at the cost of complex interaction between the tasks, such as preempting tasks evicting used data of preempted tasks out of the processor's cache. Nevertheless, some task-sets are only schedulable preemptively and, in addition to that, a non-preemptive schedule often exhibits a worse processor utilization. Thus, being able to compute both safe and precise WCET bounds for preemptive task-sets is essential.

For modern hardware architectures, however, Liu and Layland's assumption of negligible context switch costs [5] no longer holds. Instead, these costs often contribute substantially to the overall execution time, as Li et al. recently published in [4].

In this paper, we propose a new method which on the one hand decreases the context switch costs and on the other enables a precise and safe WCET analysis for preemptively scheduled tasks. Our method is an extension of the task mapping approach described in [1] which aims to increase the overall performance of a preemptive system. In our approach, we compute an arrangement of the tasks and its data in the memory such that the number of evicted cache entries of a task is minimized during preemption. The memory layout also induces a classification of the cache-entries which is then used to safely approximate WCETs under preemption. A major advantage of this approach is that both code and data remain unmodified, only the position in memory and thus in cache is changed.

The paper is structured as follows. In Section 2, we give a short intuition of our approach and the role

[1]This work was supported by the European Community's Sixth Framework Programme as part of ARTIST2 Network of Excellence and by the Seventh Framework Programme as part of PREDATOR.
[2]Universität des Saarlandes, Im Stadtwald 15, 66041 Saarbrücken, Germany
[3]AbsInt Angewandte Informatik GmbH, Science Park 1, 66123 Saarbrücken, Germany

of different memory layouts. The optimization and analysis is described in Section 3, and compared to the related work in Section 4. Finally, Section 5 concludes this paper.

2. Memory Layout

The memory layout, i.e. the arrangement of data and instructions in the memory, determines the cache-sets to which data and instructions are mapped. Therefore, it strongly influences the cache interference and thus the context switch costs of preemptively scheduled periodic tasks.

Figure 1 depicts the correlation between the memory layout and the occupied cache-sets. A task-set with three tasks of size $n/2$ is scheduled such that only task 1 can preempt the other two. The system uses a direct mapped instruction cache of size n. In the first memory layout, if task 1 preempts task 3, the task might evict cache-entries of task 3 and thus induce context switch costs. In the second memory layout, no matter which task is preempted by task 1, no cache conflicts occur. A cache-set is called *endangered*, if it might be evicted during preemption and *persistent* otherwise. This notation, however, only relates to persistence during a single instance of a task, not to different instances of it. Hereby, tasks are seen as procedures periodically invoked by the scheduler. Note that finding a memory layout such that all cache-sets are persistent during preemption is impossible in general.

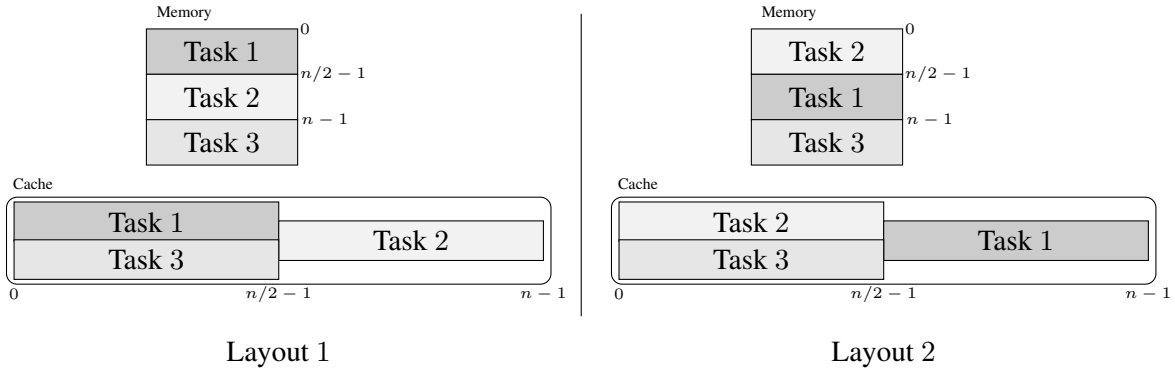

Figure 1. Two different memory layouts with different performance

3. Optimization and Analysis

In the following, we propose a combination of optimization and analysis of a memory layout in order to compute safe WCET bounds for preemptively scheduled tasks. First, we analyze the tasks to derive a metric to compare different memory layouts. We then approximate an optimal layout with respect to this metric and classify cache-entries as persistent or endangered. This classification is then used to compute safe WCET bounds for all tasks. The structure of the approach is shown in Figure 2.

In the remainder of this paper, we will use the following notation:

A cache is determined by the number of cache-sets m and the minimal life span k. The set of all cache-sets is denoted by $\mathbb{S}$. The minimal life span determines the minimum number of (read or write) accesses to a specific cache-set until the data of the first access may be evicted. This means that one can guarantee that after k different accesses, data of the first one is still cached, but after $k+1$, one can not. A direct mapped cache has $k = 1$ since the second access (to the same set with different data) removes the data of the first. A 4-way LRU cache has $k = 4$ since a cache-set can hold data

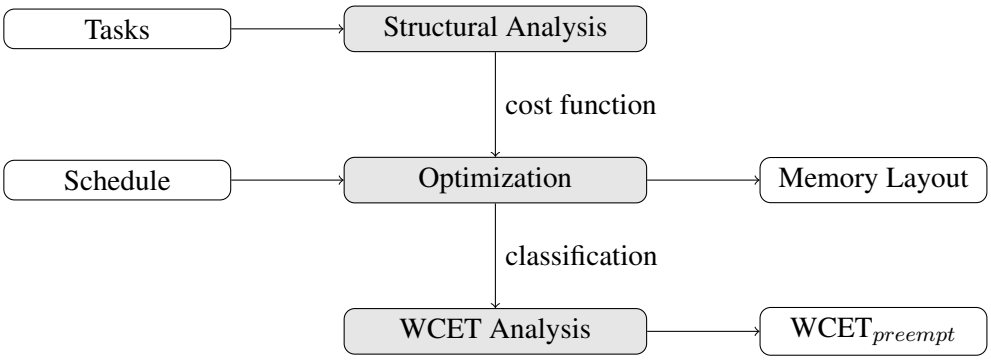

Figure 2. Overall structure of the approach

of 4 different accesses, but one further access will lead to eviction of data of the first one [7]. The analyzed hardware architecture may comprise either disjoint or unified instruction and/or data caches. Our approach is able to cope with both.

A task-set with n tasks is denoted with $T = \{\tau_1, \ldots, \tau_n\}$. The symbol τ_i denotes the task itself as well as the instructions of task τ_i. Each task has code size cs_i given in the number of cache-sets the code occupies[2]. In addition to the codesize, each task τ_i refers to a set of data fragment $D_i = \{d_{i,1}, \ldots, d_{i,l}\}$ where each such fragment has a size denoted with $ds_{i,j}$. The set of all data fragment of all tasks is denoted with $D = \bigcup_i D_i$. Data fragments refer to contiguous data blocks, arrays for instance, used by the tasks. The placement of these data blocks can be modified such that only the cache behavior (but not the semantics of program) is changed.

A task dependency relation $\vdash \subseteq T \times T$ determines the possible preemptions of the tasks. If $\tau_i \vdash \tau_j$ holds, task τ_i can preempt τ_j. Usually, the specification of communication channels or the assignment of static priorities implies such a dependency relation. For instance, with static priorities $Pr : T \to \mathbb{N}$ the relation is defined as: $\forall \tau_i, \tau_j, Pr(\tau_i) \leq Pr(\tau_j) : \tau_i \vdash \tau_j$. The relation $\vdash$ is reflexive to handle the fact that data may be evicted by other data of the same task.

A memory layout L maps code and data to start addresses in the memory. It is formally defined as

$$L : T \cup D \to \mathbb{S}$$

The start addresses are also given in the unit of cache-sets, i.e. modulo line-size. Although the function L allows empty fragments within the memory, i.e. parts which are not occupied by data or instructions, we only consider contiguous memory layouts.

The function

$$occ : (T \cup D) \times \mathbb{S} \to \mathbb{N}$$

determines how often a cache-set is occupied by a task or data fragment. For instance, if the cache has 128 sets and a task's code with size 129 starts at the first set, the first set is occupied twice (assuming usual modulo cache-mapping) and the others once. This function depends on a specific memory layout L, which we omit for the sake of simplicity of the notation.

The cost of a memory layout is determined by the possibly evicted cache-entry of all tasks. A cache-entry of a task τ_i may be evicted during preemption, if the same cache-set is occupied at least $k + 1$

[2]Note that we always refer to size as the size in number of cache-sets, i.e. $\lceil$size in bytes / size of a cache-line$\rceil$.

times by data of conflicting tasks ($\tau_j \vdash \tau_i$). Remember the definition of k. The cache can store data of k different accesses, one more access will lead to eviction.

The function $conf : (T \cup D) \times \mathbb{S} \mapsto \mathbb{N}$ defined as

$$conf(d_{i,j}, s) = \begin{cases} \sum\limits_{\tau_l \vdash \tau_i} occ(\tau_l, s) + \sum\limits_{\tau_l \vdash \tau, d \in D_l} occ(d, s) & \text{if } occ(d_{i,j}, s) > 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$conf(\tau_i, s) = \begin{cases} \sum\limits_{\tau_l \vdash \tau_i} occ(\tau_l, s) + \sum\limits_{\tau_l \vdash \tau, d \in D_l} occ(d, s) & \text{if } occ(\tau_i, s) > 0 \\ 0 & \text{otherwise} \end{cases}$$

returns the number of possible conflicts of task τ_i or data fragment $d_{i,j}$ in cache-set s.

The costs of a memory layout are thus computed by the sum over all tasks and all data fragments over all sets, where the number of conflicts exceeds k (which simply implies that these tasks or data fragments are considered endangered):

$$C = \sum_{x \in T \cup D} W(x) \left(\sum_{s=1}^{m} conf'(x, s) \right)$$

with

$$conf'(x, s) = \begin{cases} 1 & \text{if } conf(x, s) > k \\ 0 & \text{otherwise} \end{cases}$$

and a weighting function W which reflects a certain metric (as described in the following section).

3.1. Structural Analysis and Metric

The context switch costs are determined by the number of cache-sets which are 1) evicted during preemption and 2) reused by the preempted task. The memory layout shows only possibly evicted cache-sets. Which cache-set will be reused, and thus reloaded after preemption, depends on the structure of the task. If, for instance, each instruction of a task is executed at most once (during a single instance of that task), the context switch costs due to the instruction cache will be minimal. In contrast, loop structures may contribute significantly to the costs. Therefore, the pure number of evictions is not an appropriate metric to decide on an optimal memory layout. A simple metric that respects the task structure is to weight data depending on their maximal execution count.

Therefore, the structural analysis derives the following information needed for the cost function:

- size of tasks cs_i
- data fragments $d_{i,j}$ and size of data fragments $ds_{i,j}$
- weights

The size cs_i of a task τ_i can be read off the tasks directly. We employ a static analysis to derive the size of the accessed data, as follows: for a single access, the size is given as the width of the access;

for adjacent data, the accesses are combined to larger data fragments. Hereby, we assume that the targets of the memory accesses can be precisely computed (or at least overapproximated) statically.

The metric used to rate memory layouts is strongly determined by the weight function W, which is defined as follows:

$$W : T \cup D \rightarrow \mathbb{N}$$

The weight function has to be chosen such that the eviction of data with low weight has minor impact to the context switch costs than the eviction of data with high weight.

To accomplish this, we weight each data fragments in the following way:

$$W(d_{i,j}) = \begin{cases} 2n & \text{if } d_{i,j} \text{ is accessed in a loop} \\ 1 & \text{otherwise} \end{cases}$$

$$W(\tau_i) = n$$

Depending on the program structure, i.e. if a data fragment is accessed in a loop or not, the weights are assigned to the data fragments. The weight function assigns the value n to all instructions of all tasks. The number $2n$ for the data fragments is chosen to ensure that the eviction of a loop fragment weighs more than the eviction of non-recurring code or memory accesses and thus becomes much more unlikely (as previously defined, n is number of tasks). Note that the current weight function is only preliminary and still has to be evaluated. Further analyses of the structure of the tasks can be used to increase the accuracy of the metric.

3.2. Optimization

The next step is to find an optimal memory layout, or, at least, a near-optimal layout. Remember that we restrict the method to contiguous memory layouts, i.e. memory layouts without empty spaces. This means that such a memory layout is described by a sequence of tasks and data fragments: element x_i starts at the end of the preceding element x_{i-1}, i.e.

$$\forall x \in (T \cup D) : L(x_i) = L(x_{i-1}) + \begin{cases} cs_{i-1}, & \text{if } x_{i-1} \in T \\ ds_{i-1}, & \text{if } x_{i-1} \in D \end{cases}$$

Due to this restriction, all memory layouts are permutations of an initial layout. Such a permutation is denoted with the symbol σ and C_σ denotes the costs of the memory layout described by the permutation σ. A permutation σ' is a neighbor of permutation σ, iff σ' can be reached from σ by swapping the position of two elements within σ. The set of all neighbors of permutation σ is denoted by $Ne(\sigma)$.

To approximate an optimal memory layout we are using hill-climbing as shown in Figure 3: the algorithm starts with a random permutation σ_{start}. It then selects the neighbor of σ_{start} with the lowest costs and continues searching an optimal layout from this element on. In case no further improvement is possible, the algorithm may select the second best result to explore a larger portion of the state space. A predefined parameter p restricts the number of times the algorithm selects a second best permutation. By this, the parameter P can be adjusted to treat precision against running time of the algorithm. The set $visited$ keeps track of all already seen permutations to ensure that each element is visited at most once.

```
hill_climbing (permutation σ_start, unsigned int p)
{
  σ_cur = σ_start
  σ_best = σ_start
  visited = {σ_start}
  while (p > 0) {
    /* select next candidate */
    let σ' ∈ Ne(σ_cur)
    with C_σ' = min({C_σ | σ ∈ Ne(σ_cur) \ visited})
    visited = visited ∪ {σ'}
    /* is it better than the current? */
    if (C_σcur > C_σ') {
      /* is it best permutation seen so far? */
      σ_cur = σ'
      if (C_σbest > C_σ') { σ_best = σ' }
    }
    /* if not, continue with a worse result */
    else {
      p = p − 1
      σ_cur = σ'
    }
} }
```

Figure 3. Hill climbing to compute an optimal memory layout

3.3. WCET Analysis using Cache Classification

The memory layout induces a classification on all memory accesses of all tasks: a cache-entry of a task τ_i is either persistent (in case the number of conflicts is less than or equal to k) or it is endangered.

$$\text{classify(x, s)} = \begin{cases} \text{endangered} & \text{if } conf(x, s) > k \\ \text{persistent} & \text{otherwise} \end{cases}$$

This classification can be easily used to compute a safe WCET bound under preemption. If a low-level analysis detects an access to an endangered cache-set, the analysis has to handle both cases: cache-miss and cache-hit. In case of an access to a persistent cache-entry, the analysis behaves as usual. The computed WCET bound is valid, even if the task is preempted; only the endangered cache-sets might be invalidated. Cache-related timing anomalies are also treated correctly: the analysis assumes both cases and thus computes a valid WCET bound even if a cache-hit might result in a higher execution time than a cache-miss.

4. Related Work

Cache partitioning to prevent task interference during preemption has been proposed by Mueller [6] and Wolfe [14]. The cache is divided into uniform segments such that each task operates on it own. Hereby, tasks can not interfere on the cache and thus, the WCET analysis for non-preemptive systems can be used. In contrast to our method, not only the memory layout and thus the memory-to-cache mapping is adapted, but also the code itself underlies major modifications; in order to adapt to code

and data to fit into its cache segment, new branches and computation for data accesses have to be introduced. This, in addition to the highly decreased cache-size for each task, impairs the performance of the system even more.

A WCET analysis for preemptive system has been proposed by Schneider [9]. In his approach, a preemption is possible and thus assumed at every program point of the analyzed task. Thus, the analysis considers each cache access to be unknown. Obviously, the analysis computes safe approximations of the WCETs of preemptively scheduled tasks. The precision of the approach, however, suffers from this highly pessimistic assumption. Compared to our approach, we can classify cache-sets as persistent and can thus reduce this overapproximation.

The most eminent approach to timing analysis for preemptive systems computes the context switch costs separately from the WCET bound of a task. Hereby, the notion of useful cache blocks (UCBs) introduced by Lee et al. [3] plays a major role. A cache block is *useful* at a certain program point, if it may be accessed again after this point. Thus, if the cache block is evicted during preemption, the program may need to reload it and the time for this contributes to the context switch costs. Staschulat et al. [11] and Tan et al. [13] extended this approach to increase the precision (mainly by computing the set of possibly evicted cache-set in addition to the UCBs). Adapted schedulability analyses that incorporate context switch costs have been proposed in [12] for static priorities and in [2] for dynamic priorities. The main difference to our approach is on the one hand the optimization of the memory layout – which also could reduce the number of useful cache blocks – and on the other hand the handling of cache-related timing anomalies [8]. Only counting the time needed to reload cache-sets does not obey the fact that a cache-hit may lead to a higher execution time than a cache-miss. In our approach, the WCET analysis directly incorporates the cache-set classification and thus derives safe upper bound (also in the presence of timing anomalies). In addition, the usual schedulability analyses can be applied.

Other approaches, by Sebek [10] for instance, rely on measurement. However, even for a single task, full coverage is hardly achievable. Preemptive scheduling introduces an even higher level of complexity, rendering measurement-based approaches nearly infeasible – at least, no guarantee that the measured execution times deliver safe upper bounds can be given.

5. Conclusion and Future Work

In this paper, we propose a new approach to optimize and analyze the WCET of preemptively scheduled tasks. Our approach uses the fact that different memory layouts can lead to vastly different context switch costs. We first derive a metric to rate memory layouts and then approximate an optimal one. Such a memory layout induces a classification of the cache-entries into *endangered* or *persistent*. This information is then incorporated in a traditional WCET analysis allowing the analysis to derive both safe and precise worst case execution time bounds of preemptively scheduled tasks.

In the future, we plan to implement the whole toolchain and to provide an evaluation of the presented approach. To further improve the precision of the approach, a more fine-grained metric could provide more accurate cache-entry classification, e.g., by taking a maximal execution count of instructions into account. The current approach only copes with preemption occurring at arbitrary program points. Thus, an analysis of the whole schedule could provide more details about the task-set (preemption points), allowing our approach to compute a tighter set of endangered cache-entries.

References

[1] Gernot Gebhard and Sebastian Altmeyer. Optimal Task Placement to Improve Cache Performance. In *Proceedings of the 7th ACM Conference on Embedded Systems Software (EMSOFT'07)*, pages 259–268, October 2007.

[2] Lei Ju, Samarjit Chakraborty, and Abhik Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1623–1628, San Jose, USA, 2007.

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

[4] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 2, New York, NY, USA, 2007. ACM.

[5] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[6] Frank Mueller. Compiler support for software-based cache partitioning. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 125–133, 1995.

[7] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Predictability of Cache Replacement Policies. Reports of SFB/TR 14 AVACS 9, SFB/TR 14 AVACS, September 2006.

[8] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[9] Jörn Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium 2000*, pages 195–204, November 2000.

[10] Filip Sebek. Measuring cache related pre-emption delay on a multiprocessor real-time system. In *Proceedings of IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, London, 2001.

[11] Jan Staschulat and Rolf Ernst. Scalable precision cache analysis for preemptive scheduling. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 157–165, New York, NY, USA, 2005. ACM.

[12] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 41–48, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *Trans. on Embedded Computing Sys.*, 6(1):7, 2007.

[14] Andrew Wolfe. Software-based cache partitioning for real-time applications. *Journal of Computing and Software Engineering*, 2(3):315–327, 1994.