Optimal Task Placement to Improve Cache Performance

Gernot Gebhard
Saarland University
Im Stadtwald 15
Saarbrücken, Germany
gebhard@cs.uni-sb.de

Sebastian Altmeyer
Saarland University
Im Stadtwald 15
Saarbrücken, Germany
altmeyer@cs.uni-sb.de

ABSTRACT

Most recent embedded systems use caches to improve their average performance. Current timing analyses are able to compute safe timing guarantees for these systems, if tasks are running to completion. If preemptive scheduling is enabled, the previously computed timing guarantees no longer hold. At each program point, a preempting task might completely change the cache content. This observation has to be considered by timing analyses, which inevitably increases their complexity. Additionally, these cache-interferences influence the overall performance of such systems. The position of a task's data determines the portion of the cache the task will occupy, and by this, the cache-interferences of the different tasks. In this paper, we present a novel method that computes an optimal taskset placement with respect to the above criteria. This means, our method modifies the starting addresses of the tasks such that the number of persistent task sets is maximized for each task. We show that the problem of finding an optimal placement is NP-hard and present a heuristic to approximate an optimal solution. Finally, we demonstrate by means of simulations that our method is able to improve the overall performance especially of heterogeneous and complex tasksets.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*formal models, worst-case analysis*

General Terms

Algorithms, Performance

Keywords

cache analysis, predictability, task placement

1. INTRODUCTION

An embedded system usually runs several tasks. In a non-preemptive schedule, each task is running to completion. Therefore, it is possible to optimize and analyze each task independently from all others. In [3, 14] it has been shown, that such analyses can achieve safe and precise timing guarantees. The interference of the tasks can be neglected.

Preemptive systems offer a higher degree of freedom. A running task may be interrupted to ensure that a higher priority task meets its deadline. Many tasksets that are schedulable using a preemptive system (neglecting context-switch costs) are not schedulable under a non-preemptive one.

This higher variability comes at the cost of increased complexity. Timing analyses have to consider that at each program point of an analyzed task, the current cache and pipeline state may switch to a completely different one caused by preemption. Especially in the presence of timing anomalies [10], the costs caused by context switches cannot be easily predicted. But not only timing analyses for single tasks, also the performance of the whole system is subject to a more complex behavior. This complexity is mainly caused by the interference of tasks on the cache, where periods, priorities, dependencies between tasks and, of course, the memory placement influence the performance.

In this paper, we propose a new method to analyze and optimize the overall performance of an embedded system with preemptive scheduling during compile time. To achieve these goals, we modify the placement of instructions in the memory to reduce the number of cache misses and to classify instructions as persistent or as endangered with respect to preemption. This information can then be used to derive safe timing guarantees which is this far considered infeasible or imprecise in practice for preemptive scheduling. Hereby, we have to stress the fact that we do not try to ensure or enforce persistence of cache-lines up to the next occurrence of a task, but only during preemption of a task.

Our method exploits the fact that different memory arrangements lead to different performances. In a first step we collect information which might influence the performance like periods, sizes of tasks and data and so on, whereas we do not require that each task fits inside the cache. In the second step, we build an objective function out of this data and compute or approximate an optimal task distribution scheme. This scheme determines the start-address for each task's data. In the last step, we arrange the tasks according to our solution from the second step.

The code remains untouched, our approach only modifies the absolute positions in memory and herewith the positions in cache. The optimization function gives us the means to define cache-blocks of hard real-time tasks as more important than those of soft real-time tasks. Since we are completely aware of the cache behavior, we can a priori determine, which cache blocks are persistent during one run of a task and which cache blocks are not. Hereby precise timing analysis with respect to preemption becomes a reachable goal.

The remainder of this paper is structured as follows. Section 2 discusses related work. After demonstrating the influence of different task placements and introducing our assumptions about the used system and taskset in Section 3, we describe our task placement method in detail in Section 4. Section 5 evaluates our approach by means of simulations. Finally, Section 6 concludes this paper.

2. RELATED WORK

The problem to handle cache-interferences within a preemptive systems has been addressed by several authors with very diverse approaches. Targeting the problem of timing guarantees we have to mention the cache-partitioning approach by Mueller [8], the combined timing and scheduling analysis by Schneider [12] and the analysis of useful cache blocks to compute context-switch costs by Lee et al. [6].

In [8] Mueller proposed a simple method to avoid interferences of tasks on the cache by assigning each task a small segment of the cache to work with. Although there is no interference on the cache, the performance of each task will suffer by the strongly decreased available cache size. Additional branches have to be added and for data arrays, a wrapper function has to be applied. This implies significant code changes, which again in addition to the increased number of cache misses impairs the performance. Other approaches like [15] concerning cache partitioning exhibit mainly the same disadvantages.

Schneider [12] proposes a combined scheduling and timing analysis. This analysis assumes preemption at each program point to compute safe results. However, just because preemption is assumed everywhere, the analysis roughly overestimates a task's worst case execution time.

In [6] Lee et al. determine useful cache blocks to compute context switching costs. Their approach computes for each program point those cache-sets that may be accessed later and that could be evicted by an preempting task. The costs that arise from a context switch are then the time needed to reload these cache-blocks. In the presence of timing anomalies however, this approach must be considered unsafe.

For the optimization of the overall performance, we have to mention two approaches:

Suh et al. [13] set up an analytical cache model to analyze and optimize the cache-miss behavior of task sets. Given a round-robin schedule, cache-miss-rate curves for each task and the cache size, the authors are able to predict the overall cache-miss rate. Additionally, they show how their analytical cache model can be used to dynamically partition the cache. To optimize the overall cache-miss rate, the scheduler assigns cache blocks to tasks and dynamically decides to lock or unlock them, depending on the tasks cache-miss-rate curves. This approach differs from our by the use of dynamic partitioning and cache locking. Additionally, we are not restricted to a specific scheduler.

Banakar et al. [1] propose to exploit the scratchpad memory of an embedded system. The basic idea is not to rely on the cache and store frequent data in the scratchpad memory. In this fashion, the authors manage to avoid the eviction of important data, but are forced to modify the code. The authors do not address systems with preemptive scheduling. This approach is out of scope for this paper, because it forms a complete alternative to caches.

The next approach copes with a different problem but is nevertheless strongly related to our work.

In [4] Gloy et al. propose a procedure placement algorithm that minimizes the instruction cache conflicts of a single task. Their algorithm optimizes a task by means of the cache configuration, the sizes of the procedures and program traces. The program traces are used to infer conflicts between procedures of the task to optimize. The presented algorithm is designed to cope with direct-mapped caches only, but the authors provide an extension to two-way set-associative caches that use the LRU replacement policy.

Guillon et al. [5] have improved the heuristic presented in [4]. The authors found that minimizing cache conflicts by maximizing the overlap between procedures does on average not contribute to reduce cache conflicts. Making the algorithm of Gloy et al. sensitive to code size expansion, the authors manage to achieve very good results with a minimal code size expansion. However, the improved algorithm only copes with direct-mapped instruction caches.

The main difference is of course the target itself; we aim to optimize and analyze whole task sets instead of a single task only. The optimization of single tasks is again only valid for non-preemptive systems. For several reasons, we cannot adopt the former approach directly to our target: the interaction between tasks is highly dynamic. Therefore a few traces are not sufficient to capture enough data for a preemptive system. Also the analysis of endangered or persistent cache sets has not been treated sufficiently to be used for timing analyses. Finally our method is not restricted to a specific type of cache or replacement policy.

3. BASICS

Before introducing our method in Section 4, we explain how the placement of tasks in main memory influences the performance of a taskset. Figure 1 shows a taskset with three tasks, where each task occupies $n/2$ cache sets. The used cache is a direct-mapped instruction cache with n sets. The first and the third task have a short period and share their execution via time slicing, whereas the second task has a high period and thus executes very infrequently. Obviously, the second task distribution scheme leads to a much better performance than the first one, where the first task will evict the instructions from the third task whenever control switches and vice versa. The influence of the second task

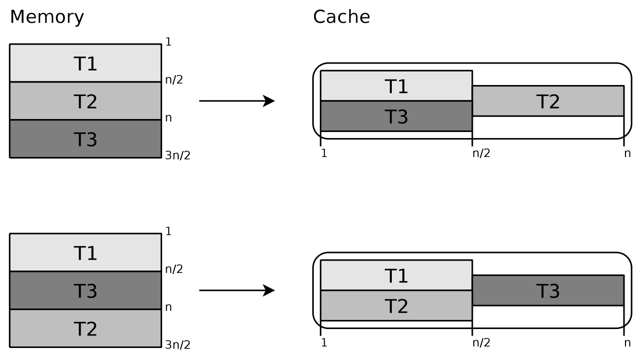

Figure 1: Two different task distribution schemes.

can be neglected because it occurs too infrequently. Naturally, this example is very simple and does not represent the common case. Nonetheless, it clearly demonstrates how the task placement influences the overall performance of a taskset.

For the sake of simplicity, we concentrate on instruction caches exclusively and ignore the tasks' data completely. Our method can be extended to data or unified caches, if the tasks do not make use of dynamic data structures. This restriction is acceptable for embedded systems, where memory is a very limited resource.

The following subsection gives an overview of our assumptions concerning the used system and taskset. Afterwards, we identify those factors that influence the performance of a taskset.

3.1 Assumptions

System

We assume an embedded system with a single processor. The architecture is supposed to possess an instruction cache using any replacement policy. In the following we explain our method by means of the least-recently-used (LRU) replacement policy, which is known to allow static analyses to predict the cache contents best [9]. As noted above, we expect a system with preemptive scheduling. Most importantly, we assume a system without virtual memory. In a system with virtual memory, we are no longer able to assign cache sets to a specific task statically. This assumption is not that restrictive because most time-critical embedded systems do not use virtual memory.

Taskset

Concerning the taskset, we do not have any restrictions. We allow a mixture of hard and soft real-time tasks. The tasks may use shared memory to communicate with each other. We expect the task-interdependencies to be known a priori, i.e. by which tasks each task might be interrupted. Note that, as discussed in Section 5, the structure of the taskset determines how successful our cache-optimization can be.

3.2 Cache Performance Factors

The cache performance is influenced by system and taskset properties, which are discussed below.

System

On the one hand, the *cache size* is of major importance. The smaller the cache and the more tasks completely fill up the cache, the less chance for optimization exists. However, as the cache sizes are steadily increasing, optimal task placements become more and more important.

On the other hand, the associativity and the replacement policy of the cache have a strong influence on the performance of the system. Both parameters determine how many different accesses are required until a certain cache-block might be evicted from the cache. For instance, if the system uses a direct-mapped cache, two different memory accesses that point to the same cache set are necessary to evict the data of the first access from the cache. If a four-way set associative LRU-cache is used, at least five different accesses to the same set are required to achieve the same effect. The maximum number of accesses that are allowed before a cache-line might be evicted from the cache is the *minimum life span* [9].

Taskset

The following taskset properties have an effect on the overall performance of the system:

task size: needed to identify the sets the task will occupy

task period: the more frequently a task occurs, the more likely it will evict data from other tasks

task-interdependencies: determine which tasks might interrupt other tasks

real-time constraints: soft and hard real-time tasks have to be handled differently

Although there are probably other factors, such as the structure of each task or the cache-miss behavior of a task [13], we only consider the above factors. Most important of the above factors are the task interdependencies. Consider two tasks a and b, where both tasks are mapped to the same cache-sets. If task a cannot preempt task b and vice versa, neither task can evict the other task's cache entries while the other task is running[1]. However, if task a may preempt task b, task a could evict data of task b from the cache, causing task b to take longer to finish execution.

4. TASK PLACEMENT METHOD

This section we present our method to compute an optimal task distribution scheme. The scheme is optimal with respect to a specific cost function that is defined below. The cost function depends on the cache performance factors we have identified in Section 3.2.

[1]Note that we only aim at keeping cache entries of a task persistent during its execution. We do not care about a task's cache entries after it has run to completion.

In the following, we formally define the input data of our task placement algorithm, which is a simple model of the cache and the task set. Then we introduce a weight function and define the notion of a task placement. A task mapping determines the set a specific task starts at. Finally, we specify the cost function to be minimized.

Definition 1. A pair $C = (m, k) \in \mathbb{N} \times \mathbb{N}$ is a *cache configuration*, where m is the cache size (i.e. the number of cache sets) and k is the minimum life span of the cache's replacement policy (see Section 3.2).

Definition 2. A *taskset* with $n \in \mathbb{N}$ tasks is a set of tasks $T = \{\tau_1, \ldots, \tau_n\}$. The subset $T_{soft} \subseteq T$ denotes the soft real-time tasks of T. The *size* of task τ_i is denoted by $S(\tau_i)$, the *period* of task τ_i is written as $P(\tau_i)$. The size of a task determines the number of cache sets the task will occupy after it is completely loaded into the cache.

Definition 3. For a given taskset T, a *task interdependency relation* is a reflexive, binary relation $\vdash \subseteq T \times T$.

Let τ_i and τ_j be two arbitrary tasks. The statement $\tau_i \vdash \tau_j$ holds, iff the task τ_i might interrupt the task τ_j during its execution and thus possibly evict cache entries belonging to τ_j from the cache. For this reason, task interdependency relations are reflexive (i.e. $\tau \vdash \tau$ for each $\tau \in T$), because a task could evict itself from the cache. This self-eviction might happen, if a task does not fit entirely inside the cache.

The task interdependency relation is used to reflect dependencies between tasks. Without this information, it must be assumed that each task can be interrupted by each other task. This, however is not true in most systems. So, for instance, if the scheduler uses static priorities, a task can only interrupt tasks with lower priorities. A second source for dependencies between tasks are communication methods or semaphores. A dependency can also be part of the task specification, where a certain task must finish before another one can start. In this case, none of these two tasks might interrupt the other. Usually, a precedence or dependency graph describes such relations. Since these dependencies are part of the specification of the system, the user has to provide the task interdependency relation defined above.

In the following, let $C = (m, k)$ be a cache configuration, T a taskset with n tasks and $\vdash$ a task interdependency relation.

We now assign an integer cost to each task. The weight of a task reflects how expensive it is to interrupt that task.[2] The weight function should assign to each task a weight that is inversely proportional to the task's period. Apart from that, the weight function should treat hard real-time tasks differently. The function is supposed to assign each hard real-time task a weight larger then the weight of every soft real-time task, to indicate that interrupting a hard real-time task and possibly evicting part of task's cache entries is the worst thing that can happen.

[2]A weight function models the performance of the task set on the cache. It can only approximately reflect the true performance.

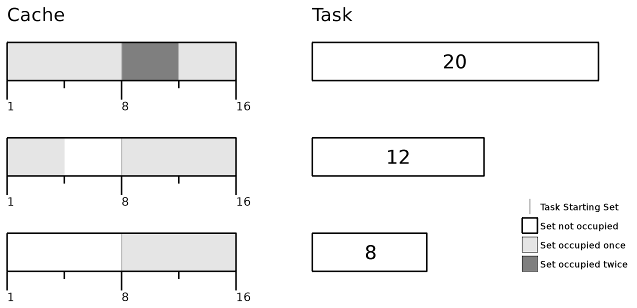

Figure 2: Shows a 16kb cache and three different tasks with different sizes. Each task starts at the same cache set.

We found that the following weight function satisfies the above requirements:

$$W : T \mapsto \mathbb{N},$$
$$W(\tau) = \begin{cases} \left\lceil \frac{\max_{i=0}^{n} P(\tau_i)}{P(\tau)} \right\rceil & \text{if } \tau \in T_{soft} \\ H \in \mathbb{N} & \text{otherwise} \end{cases}$$

where $H \gg W(\tau)$ for each $\tau \in T_{soft}$.

Although other weight functions are possible, we have seen during testing that this function produces satisfactory results.

Next, we formally define a task placement, which determines to which cache sets a task will be mapped. A task placement additionally defines how often a task will occupy a certain set. Figure 2 clearly shows how the size of a task and its placement in memory affect how often the task will occupy certain cache sets. Using this information, we can then compute the number of conflicts between two tasks.

Definition 4. A function $M : T \mapsto \{1, \ldots, m\}$ is a *task placement* of the taskset T. The $M(\tau_i)$ denotes the set, task τ_i starts at. For simplicity, we restrict the mapping such that $M(\tau_1) = 1$.[3]

The function *occ* : $(T \mapsto \mathbb{N}) \times T \times \mathbb{N} \mapsto \mathbb{N}$ computes for a given task placement how often a task occupies a certain set (see Figure 2).

Before we can define the cost function, we need to determine the number of conflicts for a given task τ and cache set i. This value measures how threatened a task's cache entry actually is. Our task placement method aims at minimizing the threat to all cache sets and thus at increasing persistence of cached data.

The number of conflicts at the set i is determined by how often the tasks preempting the task τ occupy this set, but only if τ occupies that set itself (i.e., $occ(M, \tau, i) > 0$). Otherwise, the conflicts for the task τ at the set i are zero. For a

[3]Although, we slightly restrict the solution set, we only remove redundant information. The naming of the sets does not affect the possible solutions.

given task placement M, we compute the number of conflicts as follows:

$$\mathit{conf} : (T \mapsto \mathbb{N}) \times T \times \mathbb{N} \mapsto \mathbb{N},$$
$$\mathit{conf}(M, \tau, i) = \begin{cases} \sum_{\kappa \vdash \tau} \mathit{occ}(M, \kappa, i) & \text{if } \mathit{occ}(M, \tau, i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

With the number of conflicts known for each task and set, we can finally specify the cost function in respect of which we are going to optimize the task placement. At this point, the minimum life span of the used replacement policy comes into play. If the number of conflicts at a cache set for a given task is less then or equal to the minimum life span of a cache entry, the data of that task will not be evicted from that cache set. Thus, only those sets contribute to the costs of a task placement whose number of conflicts exceeds the minimum life span.

We compute the costs of a task mapping M as follows:

$$\mathit{costs} : (T \mapsto \mathbb{N}) \mapsto \mathbb{N},$$
$$\mathit{costs}(M) = \sum_{i=1}^{n} W(\tau_i) \left(\sum_{j=1}^{m} \max(\mathit{conf}(M, \tau_i, j) - k, 0) \right)$$

The cost function is designed to reflect the true performance of a task placement. This means that for two placements M_1 and M_2 with $\mathit{costs}(M_1) < \mathit{costs}(M_2)$ the task placement M_1 should outperform M_2.

In the following section we describe two approaches to find a task placement M such that $\mathit{costs}(M)$ is minimal. In Section 4.1.1 we describe how an optimal solution can be found, by encoding the above cost function as an integer linear program (ILP). In Section 4.2.2 we discuss a simulated annealing approach, which solves a relaxed optimization problem, that is capable of finding a (near-)optimal solution much faster than an ILP solver. Additionally, we prove that both optimization problems introduced below are NP-hard.

4.1 Optimization Problem

The problem to solve is to find a task mapping M such that $\mathit{costs}(M)$ is minimal. We will denote it with the abbreviation TPP. In our first approach we used integer linear programming to solve this problem.

4.1.1 ILP Formulation

For the ILP formulation, we require two types of variables: $st_{i,j}$ and $o_{i,j}$. These variables can take the following values:

$$st_{i,j} \begin{cases} 1 & \text{if } \tau_i \text{ starts at the } j\text{-th cache set} \\ 0 & \text{otherwise} \end{cases}$$

and

$$o_{i,j} = n \in \mathbb{N}, \text{where } \tau_i \text{ occupies the } j\text{-th set } n \text{ times}$$

The variables $st_{i,j}$ can be binary variables, whereas $o_{i,j}$ must be integer variables. The latter condition is required, because a task might occupy a set several times increasing the threat to that cache set, if the task size (in terms of cache sets) is larger than the number of cache sets (i.e., $S(i) > m$).

Using these two variable types, we define the following constraints the ILP has to obey:

- Each task i has exactly one starting position:

$$\sum_{j=1}^{m} st_{i,j} = 1 \tag{1}$$

- The task i occupies the j-th set, iff it starts at most $S(i)$ sets ahead of the j-th set (remember that $S(i)$ denotes the size of task i):

$$o_{i,j} = \sum_{j'=j-S(i)+1}^{j} st_{i,(j' \bmod m)} \tag{2}$$

To complete the ILP formulation, we require two additional types of variables: $c_{i,j}$ and $c'_{i,j}$. The variables $c_{i,j}$ count the number of conflicts for the task τ_i at the j-th set (see above), whereas the variables $c'_{i,j}$ only count the number of conflicts that exceed the minimal life span k of the cache's replacement policy. To count the number of conflicts, we define the following constraints:

$$c_{i,j} \geq 0 \tag{3}$$

$$c_{i,j} \geq -B(1 - \sum_{j'=j-S(i)+1}^{j} st_{i,(j' \bmod m)}) + \sum_{\tau_{i'} \vdash \tau_i} o_{i',j} \tag{4}$$

where $B \in \mathbb{N}$ is a very large number.

The first inequality ensures that the number of conflicts is always positive. The second inequality is more complex: The latter sum simply sums over all tasks that might conflict with the task τ_i and counts how often those tasks occupy the j-th set. The first sum makes sure that $c_{i,j}$ is only positive if the task τ_i actually occupies the j-th set.[4] So, both constraints force $c_{i,j}$ to 0, if the task τ_i does not occupy the j-th set.

As mentioned above, the variables $c'_{i,j}$ only count the number of conflicts that exceed the minimum life span. We achieve this behavior with the following constraints:

$$c'_{i,j} \geq 0 \tag{5}$$

$$c'_{i,j} \geq c_{i,j} - k \tag{6}$$

where k is the minimum life span of the cache's replacement policy.

[4]To do so, we have to choose B to be at least the highest possible number of conflicts for a task.

In the last step, we have to define the objective function of the ILP. The objective function is simply the sum over the number of conflicts multiplied with the weight assigned to each task:

$$obj : \min \sum_{i=1}^{n} W(\tau_i)(\sum_{j=1}^{m} c'_{i,j})$$

4.1.2 Complexity Analysis

In this subsection, we discuss the complexity of the ILP formulation and of the optimization problem in general.

Complexity of the ILP. The main indicators for the complexity of the ILP are naturally the number of variables and constraints. For each task and cache set exist the variables $st_{i,j}$, $o_{i,j}$, $c_{i,j}$. Thus, the total number of variables is $4nm$.

Table 1 list the number of constraints of each type. The total sum is $5nm + n$.

Constraint	Number
(1)	n
(2)	nm
(3)	nm
(4)	nm
(5)	nm
(6)	nm

Table 1: Type of constraint and corresponding number of used constraints.

Although the total number of constraints and variables are in $\Theta(nm)$, the complexity of ILP grows too fast to be suitable for large task sets and caches. Instead, only small taskset can be solved within acceptable time using this approach. The complexity of the ILP however is not caused by formulation but is rather an inherent problem of TPP.

Complexity of the Optimization Problem

Theorem 1. *The optimization problem TPP as introduced above is NP-complete.*

Guillon et al. have shown the NP-completeness of a similar problem: computing a procedure placement that is optimal with respect to the number of cache misses when using a direct-mapped cache [5]. Although we handle different types of caches with arbitrary replacement policies and tasks with preemption instead of procedures, we can easily adapt the proof. Therefore we only provide a short proof sketch. For the detailed proof, we refer to Appendix 1 of [5].

Proof. To prove that TPP is NP-complete, we have to show that $TPP \in NP$ and that TPP is NP-hard.

$TPP \in NP$:
The statement directly follows by the construction of the cost function. The cost function and therefore the check for a given solution can be computed within polynomial time.

TPP is NP-hard:
To proof this claim, we have to reduce an NP-complete problem to TPP. For this purpose, we use k-colorability. The k-colorability of a graph $G = (V, E)$ can be reduced to an instance of TPP as follows: each vertex V of the graph represents one task with the length of one set and a weight of one. The conflict graph equals the original graph G. The cache has k sets and a minimal life span of 1. Each set of the cache represents one color and no two neighbored tasks are allowed to map to the same set. Therefore, if the task set can be mapped to the sets with zero costs then the graph can be colored using k different colors. □

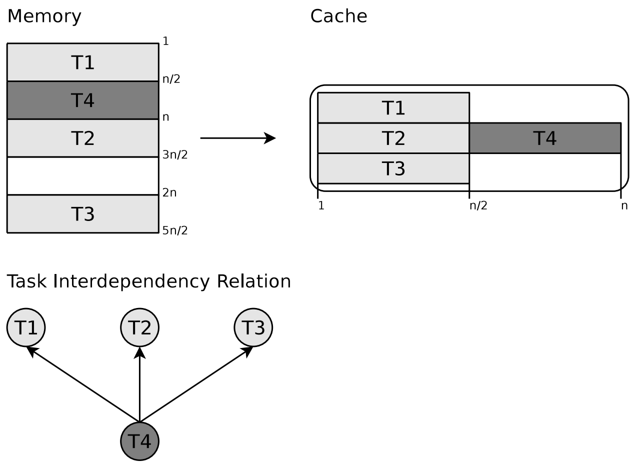

Figure 3: Optimal placement of four tasks for a direct-mapped cache comprising n sets with respect to the given task interdependency relation. Each of the tasks occupies half of the cache.

4.1.3 Remarks

We have not yet mentioned the problem of code size expansion. The starting points of the different tasks are only given modulo the cache size. So we first have to provide an algorithm that merges the data into one block. Such an algorithm was presented in [4] and [5]. In some cases, it is not possible to merge the tasks to a continuous memory block. In these cases, our method introduces empty spaces between the tasks. However, during our tests we have never encountered a task set where the best result could not be merged to a continuous block.

Figure 3 shows the smallest example for a direct-mapped cache, where a task placement with empty spaces in-between leads to smaller costs than a continuous placement. An optimal placement is achieved, if the three tasks $T1$, $T2$ and $T3$ are mapped to different cache sets than the task $T4$. Because $T1$, $T2$ and $T3$ do not interrupt each other, it is uncritical to force them to the same cache sets.

So far, we have not identified sufficient conditions under which leaving memory unoccupied leads to optimal results.

Both the code size expansion and the complexity of TPP led us to study a second approach. In practice, the ILP can only be used for small task sets and caches. So, ILP should only be used if the best solution is required. A best-effort approach is often sufficient.

4.2 Simplified Optimization Problem

The empty spaces within the memory image introduced by the first method are on the one hand very unpleasant in the scope of embedded systems, where memory is limited, and on the other hand only provide limited improvement with respect to the performance. For these reasons, we will define a second optimization problem that allows only continuous memory images. Therefore, we have investigated task permutation placements.

Definition 5. A *task permutation* $\sigma : T \mapsto T$ is a bijective function which maps each element of T to another element of the set T.

Using this definition, we can adapt the definition of a task permutation placement M'.

Definition 6. A function $M' : T \mapsto \{1, \ldots, m\}$ is a *task permutation placement* of the taskset T, iff there exists a permutation σ such that $M'(\sigma(\tau_i)) = M'(\sigma(\tau_{i-1})) + S(\sigma(\tau_{i-1}))$ for all $i \in \{2, \ldots, n\}$.

The revised optimization problem TPP' is TPP with a task permutation placement function M'.

Next, we will discuss the complexity of the revised problem TPP' and will later on provide an heuristic for it.

4.2.1 Complexity Analysis

Although we have simplified to problem to allow only continuous memory images, Theorem 2 states that the problem is still too complex to be solved exactly.

THEOREM 2. *The simplified optimization problem TPP' is NP-hard.*

PROOF. The proof bases on Theorem 1. As above, we reduce the k-colorability of a graph $G = (V, E)$ to an instance of TPP'. However, we can not use the same transformation, since the result of TPP is not necessarily a permutation. To get rid of this restriction, we have to insert dummy-tasks which are used only to fill the gaps between the other tasks. These dummy-tasks have cost and size one and have no conflict with any tasks but themself. By this, we can ensure that each task mapping forms a permutation of the task set.

The number of inserted dummy-tasks must be polynomial bounded. But since we need at most $k * |E|$ new tasks to fill all gaps the statement follows directly from Theorem 1. $\square$

4.2.2 Simulated Annealing

The restriction applied to the task placement problem shortens the solution space but as we have just shown the problem is still NP-hard. Because of this fact, we will provide an heuristic instead of an exact algorithm for TPP'.

For the heuristic approach, we used simulated annealing. The algorithm starts with a random permutation and greedily selects a neighboring permutation with the best costs.

Definition 7. A permutation σ' is a *neighbor* of the permutation σ, iff σ can be reached from σ' only by swapping the position of two tasks:

$$\exists i, j \in \mathbb{N} : \begin{array}{l} \sigma'(\tau_i) = \sigma(\tau_j) \wedge \\ \sigma'(\tau_j) = \sigma(\tau_i) \wedge \\ \forall k \in \{1, \ldots, n\} \setminus \{i, j\} : \sigma'(\tau_k) = \sigma(\tau_k) \end{array}$$

The set of all neighbors of a permutation σ is denoted by $NE(\sigma)$.

If no improvement is possible, the algorithm swaps to the second best permutation within the set of neighbors and starts searching from this new permutation on. However, selecting the second best permutation is only allowed p times, where p is a predefined parameter. The higher p is the more of the state space is visited. Of course, we have to ensure that each permutation is visited at most once. Thus, we must keep a set of visited permutation which we always substract from the set of neighbors. The algorithm in pseudo code is given in Algorithm 1.

Algorithm 1.

```
simulated_annealing (permutation $\sigma_{start}$, int $p$)
{
  $\sigma_{cur} = \sigma_{start}$
  $\sigma_{best} = \sigma_{start}$
  visited = visited $\cup \{\sigma_{start}\}$
  **while** ($p \neq 0$)
  {
    /* select next candidate */
    **let** $\sigma' \in NE(\sigma_{cur})$
    **with** $C(\sigma') = \min(\{C(\sigma) | \sigma \in NE(\sigma_{cur})\} \setminus$ visited)
    visited = visited $\cup \{\sigma'\}$
    /* is it better than the current? */
    **if** ($C(\sigma_{cur}) > C(\sigma')$)
    {
      /* is it best permutation seen so far? */
      $\sigma_{cur} = \sigma'$
      **if** ($C(\sigma_{best}) > C(\sigma')$ { $\sigma_{best} = \sigma'$ }
    }
    /* if not, continue with a worse result */
    **else**
    {
      $p = p - 1$
      $\sigma_{cur} = \sigma'$
    }
  }
}
```

5. EVALUATION

In this section we evaluate our approach. Section 5.1 discusses the framework in which we have embedded our task placement algorithm. Section 5.2 discusses the effectiveness of the heuristic presented in Section 4.2.2.

5.1 Framework

Figure 4 shows the three steps of our task placement optimization method. At first, we compile the source code into object files. Next, we determine the size of each task. For

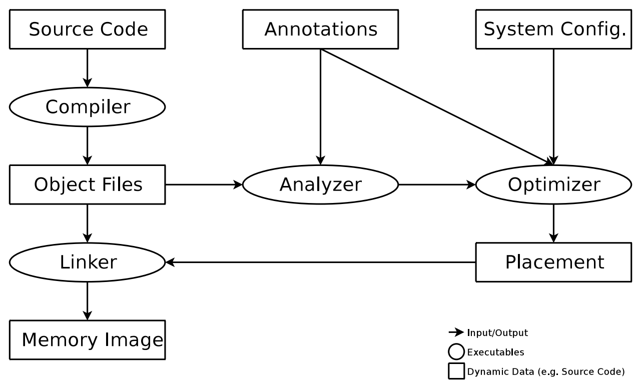

Figure 4: Task Placement Framework.

this we need annotations that indicate which functions (contained in the object files) belong to each task. Apart from that, both the task periods and the task interdependency relation are encoded in these annotations. With this information and the system configuration (amongst others, the cache configuration) we optimize the task placement with respect to the cost functions as discussed in Section 4. Finally, we instruct the linker to align the tasks according to the computed placement and to produce a memory image that can be uploaded into the embedded system's main memory.

5.2 Tests

We have evaluated our approach by optimizing several task sets with tasks from the WCET Benchmark [2]. To measure the effectiveness of our task placement method, we have simulated the tasks sets under the RTEMS operating system [11]. The operating system itself is running on a software-emulated ARM7 provided by the MPARM project [7]. To determine the effectiveness of our approach, we compared the number of cache misses of the worst and the best task placement.

Number	Task	Size (in cache sets)
1	nsichneu	1263
2	statemate	520
3	adpcm	253
4	fft1	131
5	compress	118
6	edn	115
7	lms	115
8	jfdctint	115
9	loop3	112
10	minver	108
11	SYSTEM	250
	Total Size	3100

Table 2: Shows the size in the number of cache sets for each task.

For our measurements, we have used the 10 largest tasks of the WCET Benchmark to form three different task sets. Figure 5 shows the task set interdependencies. For the sake of simplicity, we have chosen a simplified display. Instead of drawing edges between two tasks that might interrupt each other, the figure displays which tasks depend on each other.

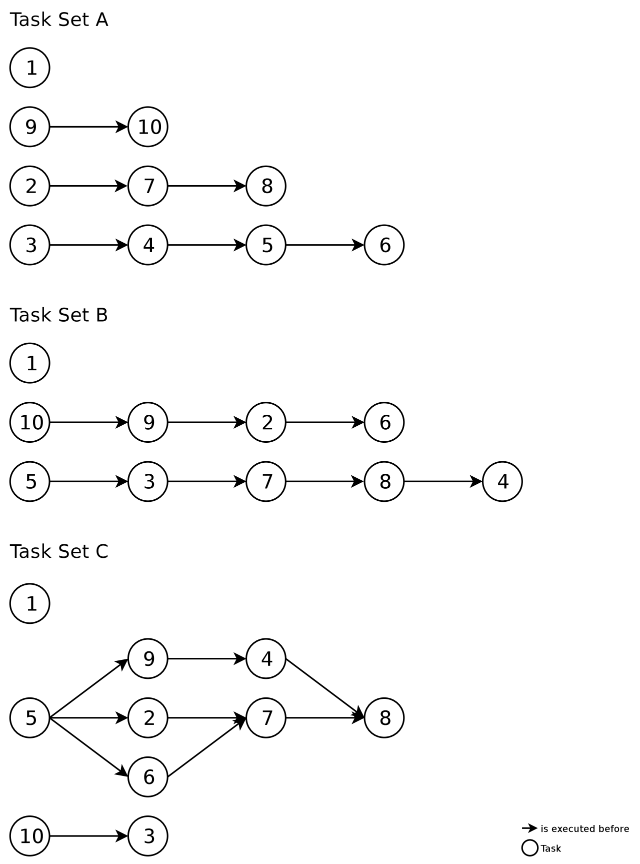

Figure 5: Shows the structure of the task sets A, B and C. Incoming edges denote which tasks have to be completed before a task may begin.

If there is no path between two tasks, they might possibly evict each other from the cache. We have designed the task sets such that task set A shows many, task set B features some and task set C offers only few possible preemptions.

We have optimized the task sets for three different instruction caches using the LRU replacement strategy:

- 16kb direct-mapped cache with 1024 sets
- 32kb two-way set-associative cache with 1024 sets
- 32kb four-way set-associative cache with 512 sets

The WCET Benchmark tasks are rather small. Thus, we have chosen small caches, to ensure that conflicts between the tasks occur. Table 2 shows the tasks and their corresponding size. The *SYSTEM* task comprises system code and is assumed to be in conflict with every other task. Note that the some tasks do not fit entirely inside the cache (divided by the associativity).

Table 3 shows the cost ratio and the measured cache-miss ratio of the best and the worst task placement for each task set and cache configuration. The measurement shows that there is no direct correlation between the number of (possible) preemptions and the decrease of cache misses. The effectiveness of the optimization strongly depends on the task set and the used cache. The ratio between the task set size and number of cache sets strongly influences the effect of our optimization. However for each test case, we were able to decrease the number of cache misses ranging from 1% up to over 50%.

However, our measurements also show that the cost ratio between the best and the worst task placement does not indicate how well the cache-miss ratio will improve. The reason for the divergence between the two ratios is to be found in the structure of the tasks. So far, we have only taken the task sizes into account and weighted each occupied cache set equally. However, certain code sections (e.g., loop bodies) are certainly executed more often. Evicting such a code section is thus more costly than evicting a code region a task visits only once. We will address this improvement in future work.

Task Set	Cache	Sets	Cost Ratio	Miss Ratio
A	direct	1024	79%	47%
A	two-way	1024	52%	62%
A	four-way	512	71%	99%
B	direct	1024	76%	85%
B	two-way	1024	52%	77%
B	four-way	512	67%	98%
C	direct	1024	81%	83%
C	two-way	1024	59%	49%
C	four-way	512	77%	98%

Table 3: Shows cost and cache-miss ratio between the best and worst task permutation placement.

Table 4 shows the percentage of persistent cache sets for the task *compress* as optimized by our analysis. In case of the direct-mapped cache, our method was not able to find a placement for any of the three task sets, where no cache set occupied by the task *compress* could be classified as persistent. To compute a safe execution time estimate for the given task, the timing analysis must assume that the content of each cache set to be unknown. In case of the two-way and four-way set-associative caches, our method was able to find a placement for every task set, where all the cache sets *compress* occupies could be classified as persistent. In this case, the previously computed WCET for that task is safe in the context of preemptive scheduling. The only exception forms the placement for task set B in case of the two-way set-associative cache, where our method was able to identify only 34% persistent cache sets. Assuming the contents of the other cache sets to be unknown, a timing analysis is then able to compute a safe as well as tight execution time estimate for the given task.

Task Set	Cache	Sets	Persistent Cache Sets	
			Best Case	Worst Case
A	direct	1024	0%	0%
A	two-way	1024	100%	0%
A	four-way	512	100%	0%
B	direct	1024	0%	0%
B	two-way	1024	34%	0%
B	four-way	512	100%	0%
C	direct	1024	0%	0%
C	two-way	1024	100%	0%
C	four-way	512	100%	31%

Table 4: Shows the percentage of persistent cache sets for the task *compress*.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a new method to optimize and analyze the cache-performance of task sets running on embedded systems with preemptive scheduling. We exploit the fact that different task placements within the memory lead to different cache-interferences during execution. For a given task set and a cache configuration our method adjusts the starting position of the tasks such that the threat to the cache sets is globally minimized. This optimization improves the cache performance on the one hand and statically classifies whether cached entries are persistent or threatened during the execution of a task. In contrast to other approaches our method does not need to modify the code and thus does not impair the execution time of a task. The cache-set classification finally makes precise timing guarantees for preemptively scheduled tasks a reachable goal.

For the optimization, we introduced a simple performance model to compare the performances of different placement and to derive the classification of the cache entries. On top of this model we developed two slightly different optimization problems and showed the NP-hardness of both. The first problem optimizes the performance independent from the used memory space and the second one only allows minimal memory usage. Additionally, we presented an ILP formulation for the first one and an heuristic approach for the second.

The measurements show that our method is able to compute cache-efficient task placements. We have observed that the effectiveness of our approach strongly depends on the optimized task set. Because we have no access to real-world applications (e.g., task sets of an automotive system), we were only able to apply our method to self-written task sets. However, we are very optimistic that our method will prove valuable in practice.

The framework we introduced within this paper offers a wide area for future work. The performance model can be enriched with more detailed information about the task set and the tasks themselves. For instance, information about recurring or sequential code segments could lead to a better approximation. To derive this information static analyses or the help of the compiler and its data could be used. Another important issue is the level of detail. Not only the tasks within the set but also the procedures of the tasks can be moved within the main memory to allow a higher variability and therefore better performances. The different steps of the method could be combined to a semi-automatic implementation of the framework where the user only has to provide the task interdependencies. Last but not least, a final evaluation of the method using task sets from industry is left for further research.

Acknowledgements

Thanks to Rheinhard Wilhelm and the anonymous reviewers for many helpful suggestions on how to improve the paper.

7. REFERENCES

[1] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign, CODES, Estes Park (Colorado)*, May 2002.

[2] Benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[3] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software Workshop*, volume 2211, pages 469 – 485, Lake Tahoe, USA, October 2001. (C) Springer Verlag.

[4] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 303–313, Washington, DC, USA, 1997. IEEE Computer Society.

[5] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez. Procedure placement using temporal-ordering information: Dealing with code size expansion. *Journal of Embedded Computing*, 1(4):437–459, 2005.

[6] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[7] MPARM. `http://www-micrel.deis.unibo.it/sitonew/research/mparm.html`.

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

[9] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Predictability of Cache Replacement Policies. Reports of SFB/TR 14 AVACS 9, SFB/TR 14 AVACS, September 2006.

[10] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[11] RTEMS. `http://www.rtems.com`.

[12] J. 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.

[13] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 1–12, New York, NY, USA, 2001. ACM Press.

[14] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separate Cache and Path Analyses. *Real-Time Systems*, 18(2/3), May 2000.

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