Cache Sensitive Pre-Runtime Scheduling

Daniel Kästner Stephan Thesing *

Universität des Saarlandes / Fachbereich Informatik
Postfach 15 11 50 / D-66041 Saarbrücken / Germany
Phone: +49 681 302 5589 Fax: +49 681 302 3065
{kaestner,thesing}@cs.uni-sb.de
http://www.cs.uni-sb.de/~{kaestner,thesing}

Abstract

We present a novel pre-runtime scheduling method for uniprocessors which precisely incorporates the effects of task switching on the processor cache into its decisions. Tasks are modelled as a sequence of non preemtable segments with precedence constraints. The cache behavior of each task segment is statically determined by abstract interpretation. For the sake of efficiency, the scheduling algorithm uses a heuristically guided search strategy.

1 Introduction

Real-time systems consist of tasks which should be finished within given time bounds. In *hard* real-time systems it even has to be *guaranteed* that the time constraints are met. Otherwise, severe consequences may result in the physical environment, controlled by the system (nuclear power plants and flight control are examples of such environments).

Hard real-time systems are often constructed as a set of periodic tasks which have an *earliest release time*, a *period* and a *deadline* associated with them. Every task must not be started before its earliest release time and must be finished before its deadline. Execution repeats after the given period.

The problem is now to find a sequence for the task execution, i.e. to determine start times for all tasks, such that all deadlines are met and earliest release times are obeyed. The problem may further be complicated by the fact that some tasks must not be run before others due to, e.g., data dependencies.

Methods to find such a sequence, which we call a *schedule*, can be classified as *runtime* and *pre-runtime* approaches. Pre-runtime schedules are computed before the system is run and provide a fixed execution sequence. Scheduling methods may further be divided into methods that allow *preemption* and methods that do not. Preemption means that the execution of a task is interrupted and another task is run followed by the resumption of the original task. This obviously extends the freedom of choosing task placements in a schedule. In the following, we focus on non preemptive pre-runtime scheduling for uniprocessors. However, we consider tasks as a sequence of *segments* and allow switching to a segment of a different task after a segment has finished its execution. A graphical example is given in Figure 1.

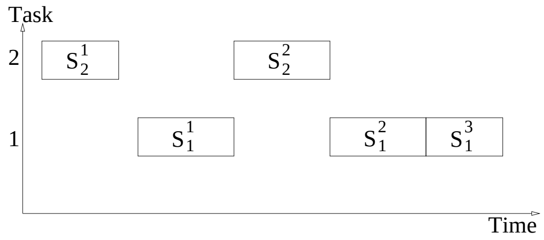

Figure 1: A task schedule with segments S_j^i

Generating a schedule requires the *worst case execution time* (WCET) of each segment to be known. If the underlying programming language is restricted, the WCET can be calculated with different methods (see [PK89, LMW95]). However, cache memories and pipelines in modern microprocessors complicate the determination of the execution time of a single (machine) instruction, as this time depends on the execution path that leads to it (see [FMW97, AFMW96, BN94, LBJ$^+$95,

*This work was done while both authors were members of the Graduiertenkolleg "Effizienz und Komplexität von Algorithmen und Rechenanlagen" (supported by the DFG).

Submitted to the 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, Montreal, CA, 19-20 June 1998.

Mue94, MWH94]). Although it is safe to count every memory access of an instruction as a cache miss, this strategy overestimates the execution time significantly[1]. In the following, we will present a pre-runtime, non-preemptive scheduling method that takes the cache behavior into account. Its WCET predictions are much tighter than that of a native scheduling method. Our method is applicable to a wide range of cache architectures. Although this paper only considers instruction caches, the method can easily be extended to data or combined instruction/data caches.

1.1 Cache Architectures

In general, we are considering an *A-way set associative cache*, i.e. the cache is divided into n/A sets, and each set is fully associative. Here, n is the capacity of the cache, i.e. it describes how many different memory blocks can be stored in the cache, and A is the number of elements in one set. This includes as special cases direct mapped and fully associative caches. We are employing a *least recently used* (LRU) replacement strategy for each set (see figure 2).

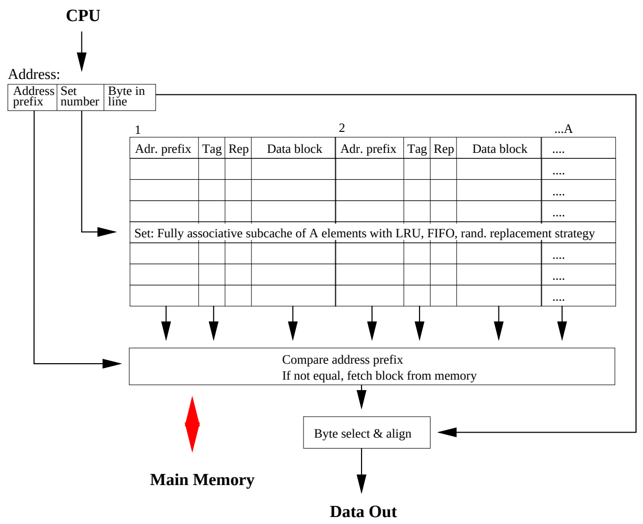

Figure 2: A-way set associative cache architecture

1.2 Determining Cache Behavior

In [Fer97] and [FMW97] it was shown how *abstract interpretation* can be used to statically determine the cache behavior of a program. Abstract interpretation [CC77] is a general framework for static program analyses. The main idea is to replace the concrete semantics operating on concrete values by an abstract semantics operating on abstract values. If the abstract versions of the concrete values and semantic functions satisfy certain conditions, a provably correct static analysis of a program can be constructed[2]. Based on this theory, a powerful tool, PAG (Program Analyzer Generator), has been constructed, which is used to generate the analyses described in this paper. PAG takes a concise specification of the analysis to be performed and generates an efficient program analyzer in the form of several **C** files that can be compiled and linked to the program in which the analysis is to be integrated. In order to perform the analysis, all that is needed is a call to the appropriate **C** function, which gets as an argument the control flow graph of the program to be considered. See [TMLA97] for details.

The analyses in [Fer97] determine a classification of every memory access of a program as

- *always hit* (ah), if the access is guaranteed to be a cache hit,
- *always miss* (am), if the access is guaranteed to be a cache miss and
- *not classified* (nc), if the access cannot statically be determined to be a hit or a miss.

The analyses associate *abstract cache states* with every program point. The abstract cache state is the abstraction of concrete cache contents and describes a set of possible cache states that may occur at that point during the execution of the program. For our needs, we will treat the above mentioned analysis as a function, taking an abstract start cache state and the control flow graph (CFG) of a segment and returning a function, which maps the nodes of the CFG to the set {ah, am, nc}, according to the previously mentioned classification. With this classification, we can compute the WCET of the segment, which is now dependent on the abstract start cache state, by using one of the methods from [PK89] or [LMW95]. Since the analysis has to be conservative, each nc has to be accounted for as an am when calculating the WCET. Note that the nodes of a CFG may be annotated with additional information, e.g. the loop bounds from [PK89]. In Figure 3, an example CFG and the classifications of a segment are shown. In the example, a cache with 16 byte line size is used; each instruction is 4 byte wide (SPARC assembler code).

[1] The access time for a cache hit is mostly 1 clock cycle and can be as high as 8-66 cycles for a miss [HP96].

[2] We are not dealing with the details of abstract interpretation here, see [CC77] or, with respect to cache memories, [Fer97] for details.

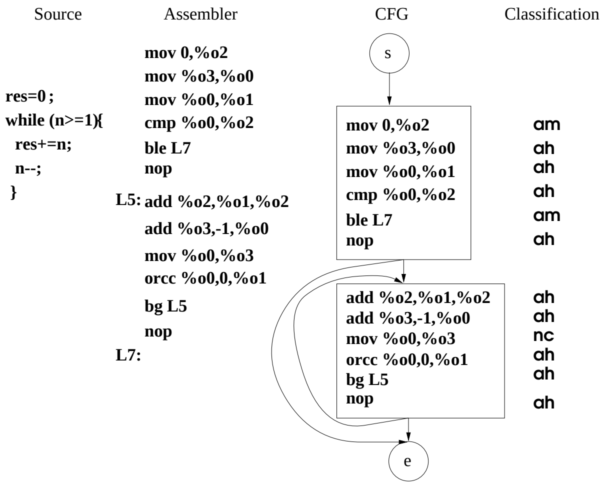

Figure 3: CFG and classification of a segment

1.3 Related Work

In [BN94], it is pointed out that the interaction of different tasks on the cache, the so-called extrinsic cache behavior, can have considerable effects on the overall cache performance. An approach that tries to incorporate cache-related preemption delays into schedulability analysis in the context of preemtive scheduling was presented in [LHYM$^+$96]. There, Lee et al. extend the response time approach of [JP86, TBW94] by including an upper bound of the costs caused by cache reloading of preempted tasks into the recurrence equation for the response time R_i of task τ_i:

$$R_i = C_i + \sum_{\tau_j \in \ hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + PC_i(R_i) \qquad (1)$$

Here, C_i is the WCET of a task τ_i, T_j its period and the set $hp(i)$ contains the tasks with higher priority than τ_i. $PC_i(R_i^k)$ denotes the preemption delay, which has to be considered when calculating the response time. This additional delay accounts for reloads of cache blocks during the execution of τ_i, that were caused by the preemption of τ_i by tasks from $hp(i)$.
To ensure the schedulability of a system, the response time of each task must be less than or equal to its deadline. The response times can be calculated by iteratively solving the recurrence equation (1):

$$\begin{aligned} R_i^0 &= C_i && (2) \\ R_i^1 &= C_i + \sum_{\tau_j \in \ hp(i)} \left\lceil \frac{R_i^0}{T_j} \right\rceil C_j + PC_i(R_i^0) \\ &\vdots \\ R_i^{k+1} &= C_i + \sum_{\tau_j \in \ hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j + PC_i(R_i^k) \end{aligned}$$

The iteration stops, when $R_i^m = R_i^{m+1}$ for some m.
At each iteration step, the values $PC_i(R_i^k)$ are calculated in two passes. In the first pass, Lee et al. determine for every program point the set of *useful memory blocks*, i.e. memory blocks that may be used again in the execution of the task beyond that point. The set of useful memory blocks can be determined by combining the results of two data flow analyses, *live memory blocks* and *reaching memory blocks*. These are derived from the well known *live variables* and *reaching definitions* analysis: a memory block m is said to be *alive* at a program point p for a cache set c, if it may be the first reference to cache set c after the program point p. It is said to *reach* p, if it may reside in cache set c at program point p. A memory block that is alive at p and reaches p is called *useful*. Useful memory blocks may be in the cache and may be used later.
The preemption cost at a program point p may then be given by the number of useful memory blocks at p multiplied by the cache miss penalty time. This corresponds to the time required to reload the useful memory blocks into the cache, if the task is preempted at p. However, since a program point may be executed several times (e.g. the bodies of loops), the costs of each program point have to be considered e times, where e is the number of executions of p. From this, a table can be constructed that gives for every task the costs in decreasing order.
In the second step, the equations (1) are solved, where $\mathrm{PC}_i(R_i^k)$ is determined by the cost table and the number of preemptions of the task by other tasks. For the computation of the number of preemptions, an integer linear program has to be formulated and solved in every iteration step.
In [LHYM$^+$96], only direct mapped caches are considered; moreover, their approach requires to solve an integer linear program at each iteration step. Since the computation of a single integer linear program is already $\mathcal{NP}$-hard in the worst case, this approach will be limited only to small task sets.
A similar approach for considering the extrinsic cache behavior in preemptive real-time systems is presented in [BMSO$^+$96]. Here, the preemption delays are modeled in a simpler way: either the time to refill the entire cache or the time to refill the lines misplaced by the preempting task is considered. The equation (1), however, is extended to incorporate the blocking time for a task, based on the Ceiling Semaphore Protocol [SRL90].
An algorithm for the problem of pre-runtime scheduling for processes with precedence and exclusion relations was presented in [XP90]. The algorithm starts with a

heuristically determined schedule that is obtained with the earliest-deadline-first strategy. This initial schedule is improved on until an optimal schedule is found. However, the cache behavior of the tasks is not taken into account.

2 Basic definitions

2.1 Scheduling

We are considering a set $\mathcal{T}$ of N *tasks* $\mathcal{T} = \{\tau_i | \ 1 \leq i \leq N\}$. Every task consists of one or more *segments*, which are executed sequentially. The i-th segment of τ_j is annotated as $\mathcal{S}_j^i$, $1 \leq i \leq \eta_j$, where η_j is the number of segments of τ_j. The set of all segments is denoted by $\mathcal{S}$.

For every τ_j, D_j denotes the *deadline* and R_j the *earliest release time* of the task. In addition we assume that there is a precedence relation $<_p$ between segments: $\mathcal{S}_j^i <_p \mathcal{S}_{j'}^{i'}$, iff $\mathcal{S}_j^i$ must not be run after $\mathcal{S}_{j'}^{i'}$. $<_p$ has to be irreflexive, transitive and anti-symmentric. For the segments of a task τ_j, $<_p$ must satisfy $\mathcal{S}_j^i <_p \mathcal{S}_j^{i+1}$, $\forall 1 \leq i \leq n_j - 1$, i.e. the segments must execute sequentially. It may be noted that a system of periodic tasks can always be transformed into the before mentioned form by considering an interval, which is the least common multiple (LCM) of the task periods, by creating LCM/p_j copies of each task with adjusted start times, where p_j denotes the period of the original task.

By cfg_j^i, we denote the (annotated) *control flow graph* of $\mathcal{S}_j^i$. $start_j{}^i$ and $end_j{}^i$ denote the (unique) start and end node of cfg_j^i.

The *scheduling interval* $P = [0, t_{\max}] \subseteq \mathbb{Z}$ is the time interval that contains one complete execution of a schedule (we consider clock cycles as basic unit of the execution time). In most cases, $t_{\max}$ will be the least common multiple of all periods of the original problem.

A *schedule* is a mapping from $\mathcal{S}$ to its starting time in the scheduling interval P. We represent T as a mapping $T : \mathcal{S} \rightarrow P$, where $T(\mathcal{S}_j^i) = t$ means that $\mathcal{S}_j^i$ starts execution at time t. The start time of τ_j is written r_j and given by $r_j = T(\mathcal{S}_j^1)$. By $C^T(\mathcal{S}_j^i)$ we denote the worst case execution time of $\mathcal{S}_j^i$ under the schedule T.

A *valid* schedule is a schedule that obeys the following restrictions:

- $\forall \mathcal{S}_j^i : T(\mathcal{S}_j^i) + C^T(\mathcal{S}_j^i) \leq t_{max} + 1$
- $\forall \mathcal{S}_j^i : \nexists \mathcal{S}_{j'}^{i'} : T(\mathcal{S}_j^i) \leq T(\mathcal{S}_{j'}^{i'}) < T(\mathcal{S}_j^i) + C^T(\mathcal{S}_j^i)$
- $\forall \tau_j : R_j \leq T(\mathcal{S}_j^1)$
- $\forall \mathcal{S}_j^i, \mathcal{S}_{j'}^{i'} : \mathcal{S}_j^i <_p \mathcal{S}_{j'}^{i'} \Rightarrow T(\mathcal{S}_j^i) < T(\mathcal{S}_{j'}^{i'})$

The first condition forces the completion of all segments before the end of P. The next condition ensures that at most one segment is executed at any given time. The third condition states that tasks must not be started before their release times. The last condition demands that a valid schedule obeys the given precedence relation.

A *feasible* schedule is a valid schedule under which all deadlines are met, i.e. $\forall \tau_j : T(\mathcal{S}_j^{n_j}) + C^T(\mathcal{S}_j^{n_j}) \leq D_j$.

Finally, a *partial* schedule T^p is a valid schedule for a subset $\mathcal{S}' \subseteq \mathcal{S}$ (with $<_p$ restricted to $\mathcal{S}'$). A segment $\mathcal{S}_j^i$ is called *scheduled* (in T^p), if $\mathcal{S}_j^i \in \mathcal{S}'$; otherwise, it is called *unscheduled.* A task τ_j is called *unscheduled*, iff $\mathcal{S}_j^1$ is unscheduled. It is *(completely) scheduled*, iff $\mathcal{S}_j^{n_j}$ is scheduled. If $\mathcal{S}_j^1$ is scheduled but τ_j is not completely scheduled, it is called *partially scheduled.*

Note that for given T, a mapping $T' : P \rightarrow \mathcal{S} \cup \{\text{Idle}\}$ can be constructed, mapping any time $t \in P$ to either a segment that is executed at t or Idle, meaning that the processor is idle:

$$T'(t) = \begin{cases} \mathcal{S}_j^i & \text{, if } \exists \mathcal{S}_j^i : T(\mathcal{S}_j^i) \leq t < T(\mathcal{S}_j^i) + C^T(\mathcal{S}_j^i) \\ \text{Idle} & \text{, otherwise} \end{cases}$$

2.2 Cache Memories

An A-way set associative cache with n lines consists of a sequence of (fully associative) sets. The store $M = \{m_1, \ldots, m_S\}$ consists of a set of memory blocks, whose addresses are determined by a function $adr : m \rightarrow \mathbb{N}$. The set, in which a memory block is cached, is determined by a function $set : M \rightarrow F$, where $set(m) = f_i$, iff $i = adr(m)\%(\frac{n}{A}) + 1$. An element of a (fully associative) set where a memory block may be started is called set line. An unused set line is assumed to contain a designated element ϵ and we define $M' = M \cup \{\epsilon\}$.

A function $\iota : L \rightarrow M'$, which maps each set line to the memory block stored in it, is called a concrete set state. Let I denote the set of all concrete set states. A concrete cache state $c : F \rightarrow I$ assigns a concrete set state to each set in the cache. Then, an abstract set state describes a set of concrete set states and can be represented as a function $\hat{\iota} : L \rightarrow \mathcal{P}(M')$. If we denote the set of all abstract set states by $\hat{I}$, we can represent an abstract cache state as a function $\hat{c} : F \rightarrow \hat{I}$ which assigns an abstract set state to each set in the cache. In the following we will use C to refer to the set of all cache states and $\hat{C}$ to refer to the set of all abstract cache states.

The MUST-Analysis, presented in [Fer97], determines a set of memory lines which are guaranteed to be in the cache, when the control flow reaches a given program point. In order to simulate the effect of a memory reference on the abstract set and cache states, appropriate

update functions have to be defined. Since set states as well as abstract set states depend on the associativity and the replacement strategy of the cache – assumed to be the LRU-strategy – we have to consider the relative age of each memory block in a set line.
The relative age of a memory block m in an abstract set state $\hat{\imath}$ must always be an upper bound on the relative ages of m in the concrete set states represented by $\hat{\imath}$. The relative age of a memory block m_a in the set can only be changed by references to memory blocks m_b, where $set(m_a) = set(m_b)$. Furthermore, memory block m_b must as least as old as m_a for the relative age of m_a to be changed.
These changes are modelled by an abstract set update function $\hat{U}_{\hat{I}} : \hat{I} \times M \rightarrow \hat{I}$, which describes the new abstract set state for a given abstract set state and a referenced memory block. The new abstract cache state is calculated by an abstract cache update function $\hat{U}_{\hat{C}} : \hat{C} \times M \rightarrow \hat{C}$ for a given abstract cache state and a referenced memory block. The update of a concrete fully associative cache for a referenced memory block s is shown in Figure 4. Two cases can be distringuished: in the first part of the figure, s enters the cache, in the second one, it is already contained in the cache and only its age has to be updated. The update of an abstract cache state for a referenced memory block s is shown in Figure 5.

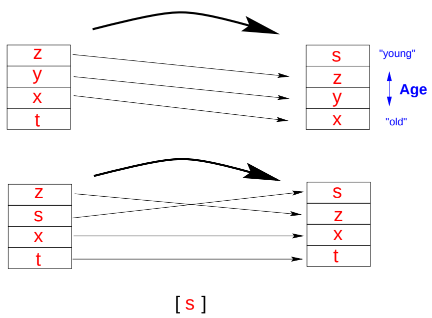

Figure 4: Update of a concrete fully associative sub-cache

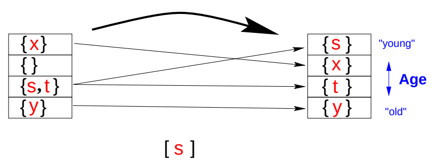

Figure 5: Update of an abstract fully associative sub-cache

When control flow joins exist in a program, abstract cache states have to be combined. A memory block only remains in an abstract set state, when it is contained in the abstract state of both operand set states. When these have different ages, the memory block is assigned to the oldest one. This is modeled by a join function for abstract set states. A join function for abstract cache states applies the join function for abstract set states to all corresponding set states of its arguments. An example of an abstract join is given in Figure 6.

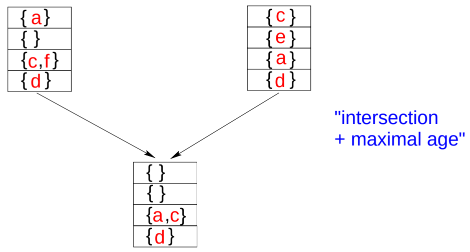

Figure 6: Abstract join for the MUST analysis

In order to solve the MUST-Analysis for a given segment, a system of recursive equations is created from its control flow graph. The variables in the equation system represent abstract cache states for all program points. The least fixed point of this equation system[3] is computed by fixed point iteration; it maps every node in the CFG to an abstract cache state.
The approximated fixed point $\mathcal{F}_{\hat{c}}(\mathcal{S}_j^i) : CFGNodes \rightarrow \hat{C}$ is computed relatively to the abstract cache state $\hat{c}$ at the start of the segment $\mathcal{S}_j^i$. This cache state $\hat{c}$ is given by $\mathcal{F}_{\hat{c}'}(\mathcal{S}_{j'}^{i'})(end_{j'}^{i'})$, if $\mathcal{S}_{j'}^{i'}$ is the segment executed before $\mathcal{S}_j^i$, or by the empty abstract cache state for the first segment in the schedule.

3 Overview

We have integrated the static cache analysis into a pre-runtime scheduling algorithm, using a depth first search technique. The algorithm is based on a branch-and-bound algorithm which was presented in [BFR71]. There the objective is to minimize the execution time of a set of tasks with given release times and deadlines. In our work, we extend this approach by considering precedence constraints and switching costs due to the inter-task cache interference. Since we have to deal with periodic tasks, we don't want to determine a feasible schedule with minimal execution time; instead we return the first feasible schedule found. Thus, the algorithm can be considered as a heuristically guided search where the search tree can be pruned under certain circumstances.
The schedule is computed incrementally. The root of the search tree corresponds to the empty schedule and

[3] Which is computable for this analysis.

the nodes of level k represent all valid schedules consisting of exactly k task segments. Clearly, the maximal depth of this tree corresponds to the number of task segments and without precedence constraints and without pruning, all $|\mathcal{S}|!$ possible schedules are enumerated in the worst case. At each node n_μ, a *segment ready set* SRS_μ is computed which contains all yet unscheduled segments that can be appended to the schedule represented by n_μ. To be more precise, SRS_μ contains exactly those task segments, which are not contained in the partial schedule and whose predecessors with respect to the precedence relation have already been scheduled. A child of a node n_μ in the search tree is created for each task segment in SRS_μ which can be appended to the schedule corresponding to n_μ without exceeding its deadline.

If the completion time of the newly added task segment $\mathcal{S}_i^j$ exceeds its deadline, then the tree can be pruned at node n_μ. The reason is the fact that, if $\mathcal{S}_i^j$ exceeds its deadline when appearing at the k-th position of the schedule, it will certainly miss its deadline when scheduled at an even later position.

If at a node on level k in the search tree, the schedule length is less than or equal to the smallest release time among the unscheduled tasks, the problem decomposes at that node. The best schedule for the remaining $n-k$ task segments may not be started prior to the smallest release time among those tasks. Thus, there is no need to backtrack beyond level k.

The order in which the children of a node in the search tree are visited can be determined by several heuristics. In our implementation, we have considered the earliest-deadline-first heuristic.

This approach allows us to consider cache effects in a straightforward way. At each node, the abstract cache state at the end of the partial schedule is calculated. Each instruction is classified as either a cache miss or a cache hit. Thus, the worst case execution time of the task segments can be calculated taking their cache behavior into account. Since the abstract cache state at the end of the parent node in the search tree is used as initial cache state, the switching costs of different task sequences are considered.

4 Scheduling Algorithm

In this section, we present our scheduling algorithm in more detail. It can be described mainly by two procedures: the procedure SCHEDULE and a recursive function RECSCHEDULE. These functions are given in a PASCAL-like pseudocode-notation as follows:

```
SCHEDULE(TASKSET p, SPECIFICATION spec)
begin
  Forall S ∈ p do
    cfg_S = CreateControlFlowGraph(S);
    InitDataFlow(cfg_S);
  od
  ReadTaskSetSpec(spec);
  l = CreateReadyList(p);
  schedule = ∅;
  Forall nodes n ∈ l do
  begin
    status = RecSchedule(n);
    if status == true then
      print schedule;
      return;
    fi
  end
  return("No feasible schedule found");
end
```

```
int RECSCHEDULE(n)
begin
  Append n to schedule;
  if length(schedule) == N then
    return true;
  fi
  Forall S ∈ SRS(n) do
    if ReleaseTime(S) ≤ EndTime(n) then
      n' =CreateNode(S)
      n' →cache = CalcCacheBehavior(S, n → cache)
      n' →lastWcet = CalcWcet(S, cl);
      if (Endtime(n)+n' →lastWcet≤Deadline(S) then
        append n' to Childlist(n);
      else
        remove n from schedule;
        return BACKTRACK;
      fi
    fi
  od
  if Childlist(n) == ∅ then
    Forall t ∈ SRS(n) with ReleaseTime(S) > Endtime(n)
      in increasing order of release time do
      n' = CreateNode(S);
      n' → root = true;
      n' → cache =CalcCacheBehavior(S, n → cache);
      n' →lastWcet = CalcWcet(S, cl);
      if (ReleaseTime(S) + n' →lastWcet≤Deadline(S) then
        append n' to Childlist(n);
      else
        exit("No feasible Schedule can exist");
      fi
    od
  fi
  Sort Childlist(n) by heuristic h;
  Forall n' ∈ChildList(n) do
    status = RecSchedule(n');
    if status == BACKTRACK then
      if n → root == true then
        exit("No feasible Schedule can exist");
```

```
        else
            remove n' from schedule;
            return false;
        fi
    fi
    if status == false then
        remove n' from ChildList(n);
    else
        return true;
    fi
  od
  if status == false and n → root == true then
    exit("No feasible Schedule can exist");
  else
    return false;
  fi
end
```

First, the control flow graphs of all task segments are created and the data flow values associated with each node are initialized. Then the specification of the actual task set is read in. This specification indicates for each task segment $\mathcal{S}_i^j$ its release time, deadline and the set of all task segments $\mathcal{S}_\mu^\nu$ which are directly dependent of $\mathcal{S}_i^j$ with respect to the precedence relation. Nodes are created for each task segment, the schedule can start with; these are inserted into a list l. Each node represents a partial schedule; it contains information about the last task segment in the corresponding schedule, the abstract cache state at the end of it and its end time. For all created nodes, the set $SRS(n)$ is calculated. This set contains all unscheduled segments that could be appended directly at the end of the schedule represented by n without violating any precedence constraints. For the nodes in l, the cache behavior prediction (see below) is performed starting from an empty cache state. The schedule is implemented as a list; it represents the path in the search tree from the root to the actual node. At the start of the algorithm, it is empty.

The heart of the algorithm is the recursive function RecSchedule; it considers the nodes in the list l one after another as starting point until a feasible schedule is found. The node n passed as a parameter is appended to the actual schedule *schedule*. If all segments have been scheduled, the function returns *true*. For each task segment $\mathcal{S}_i^j$ in the set SRS whose release time does not exceed the endtime of the actual schedule $schedule_n$, the following actions are performed: A new node n' is created which represents the schedule obtained by appending $\mathcal{S}_i^j$ to $schedule_n$. The cache behavior of that task segment is predicted starting with the cache state at the end of $schedule_n$ and the new abstract cache state at the end of $\mathcal{S}_i^j$ is saved. This is done by the function CalcCacheBehavior, which implements the static cache analysis as described in Section 2. Each instruction is either classified as a cache hit or a cache miss; the abstract cache state at the end of the actual task segment $\mathcal{S}_i^j$ is returned.

Then the worst case execution time of $\mathcal{S}_i^j$ is calculated using the cache behavior prediction performed by CalcCacheBehavior. If $\mathcal{S}_i^j$ is guaranteed to complete before its deadline, the new information is registered in n' and n' is inserted into the childlist of n. Each node in *Childlist(n)* represents a partial schedule where a task segment $\mathcal{S}_\mu^\nu \in SRS(n)$ is executed after the segments in $schedule_n$ without exceeding its deadline. If any task segment exceeds its deadline, then no feasible schedule can be found starting from node n. Then RecSchedule returns the value *BACKTRACK*. That means that we can stop considering node n and proceed with the next child in the childlist of the parent of n.

If *Childlist(n)* is empty, no task segment can be appended to $schedule_n$ without inserting idle time. Then the task segment with the earliest release time that is contained in *SRS* and does not exceed its deadline is appended to $schedule_n$. The flag $n' \to root$ is set to *true* to indicate that no backtracking beyond that node need be performed.

Subsequently, the childlist is sorted by increasing deadlines of the corresponding task segments. This way, the earliest deadline first strategy is incorporated into our algorithm, however other heuristics can be used, too. For the nodes in the childlist, the function RecSchedule is called.

No feasible solution is possible, when the return value of RecSchedule for any of the children is *BACKTRACK* and the root-flag is set to *true*. Otherwise, the actual incarnation of RecSchedule returns *false*. Then the parent of n removes n from its childlist and proceeds with the next child. A feasible schedule has been found, when RecSchedule(n') for a child n' of n returns *true*; then this return value is propagated and in the function Schedule, the generated schedule is printed out. If the return value of all children was *false* and the root-flag of n is *true*, again no feasible schedule can exist and the calculation stops; otherwise the function returns *false* and the parent can proceed with its next child.

5 Experiments

For our experiments, we reused a **C** program collection, made available by the Real-Time research group of the Seoul National Univeristy, Korea[4]. The collection contains 17 programs of 42 to 3224[5] SPARC instructions. We extracted 18 test-functions from the set (see Table 1) and randomly selected calls to these test-functions

[4] See `http://archi.snu.ac.kr/realtime/benchmark/index.html`.
[5] compiled by `gcc -O6`

Functionname	Description
`encode`	CCITT G.722 ADPCM (Adaptive Differential Pulse Code Modulation) algorithm
`decode`	
`binary_search`	Binary search program
`icrc`	CRC (Cyclic Redundancy Check) example program
`fft1`	FFT (Fast Fourier Transform) using Cooly-Turkey algorithm
`fft_c`	FFT (Fast Fourier Transform) for 1K array of complex numbers
`fib`	Fibonacci series function
`fir_filter`	FIR filter with Gaussian number generation
`insertsort`	Insertion sort
`jpeg_fdct_islow`	A JPEG slow-but-accurate integer implementation of the forward DCT
`lms`	LMS adaptive signal enhancement
`ludcmp`	LU decomposition algorithm
`matmul`	Matrix multiplication
`minver`	Matrix inversion algorithm
`sort`	non-recursive quick sort algorithm
`qurt`	Root computation of quadratic equations
`select`	N-th largest number selection algorithm
`sqrt`	Square root calculation

Table 1: Test-function set

to form the segments of a task set as follows: every segment is constructed as **C** code of the form

```
void  Seg_i_j(void)   {
  glue code
  f(arguments)
  glue code
  }
```

The glue code contains declaration of variables, initializations, etc. f is one of the test-functions from the SNU collection, i is the task number and j is the segment number.
The random selection was parameterized by the number of tasks to produce, the minimal and the maximal number of segments in a task. Another randomized parameter indicates the number of occurrences of a task within the scheduling interval to be considered; the deadlines and earliest release times have been chosen appropriately.
The generated **C** code together with the code of the test-functions has been completed to form a single executable. This executable was given as input to our scheduler. The scheduler contains the cache analyzer, as generated by PAG (see 1.2). As described in Section 4, the analysis of a segment is simply performed by a call to the analysis function, with the actual task segment and the start cache state as arguments. The resulting cache state and the classification (`ah,am,nc`) for the code in the segment is returned by this function. Since we did not dispose of an implementation of the WCET determination algorithms of [PK89, LMW95], we have used the `qpt2` program tracer of the WARTS toolkit in order to determine the execution counts of each machine instruction in a segment. To obtain timing information, we used a simplified model. We assumed that every machine instruction that was classified as a cache hit takes one clock cycle to execute, whereas each miss or unclassified instruction takes 10 clock cycles. This is a realistic assumption for current RISC-architectures; in fact, for most architectures, the difference is even greater [HP96]. In order to obtain the WCET time of a segment, we combine the results of the classification of the cache analysis with the trace of that segment. The trace of each call to a test-function is static, as we have used static arguments. This way, an upper bound on the WCET is provided.
We have performed three scheduling runs on each task set:

1. The scheduling method proposed in this paper (C).
2. Scheduling is performed by the same method, but the empty cache is used as start cache for every cache analysis (I).
3. Finally, we completely ignored all cache effects during scheduling by considering every instruction fetch as a cache miss (N).

Up to now, we are in initial stages of experimental evaluation. We have considered three randomly generated task sets for three distinct cache configurations where the number of recurrences of each segment is low. For the final version of this paper, more complete experimental results will be available. Taskset 1 contains the following functions of the test set described in 1: `matmul, jpeg_fdct_islow, sort, sqrt, fib, fir_filter, ludcmp, insertsort, minver`. In taskset2, the functions `ludcmp, fft1, fir_filter, qurt, binary_search, matmul, select, decode` are called; taskset 3 contains the functions `encode, lms, qurt, sqrt, fft1, jpeg_fdct_islow`. Each function may be called more than once during execution. The results gathered so far are summarized in Table 2. There, the column TS identifies the taskset, CS denotes the cache size (given in KB), A the associativity and LS the line size (given in bytes). The calculation mode is spec-

TS	CS	A	LS	MD	L
1	0.5	1	16	C	49734
				I	49851
	4	1	16	C	38583
				I	38952
	4	4	16	C	38655
				I	38952
	–	–	–	N	–
2	0.5	1	16	C	13766
				I	13910
	4	1	16	C	12740
				I	13379
	4	4	16	C	12731
				I	13217
	–	–	–	N	64310
3	0.5	1	16	C	43870
				I	44086
	4	1	16	C	30424
				I	31918
	4	4	16	C	30568
				I	31882
	–	–	–	N	–

Table 2: Experimental Results.

ified in column MD and the length of the constructed schedule in machine cycles is given in the last column. For the sake of simplicity, the deadlines and release times of the task segments have been chosen so, that no idle times need to be inserted. When the cache behavior is neglected, i.e. when all instructions are assumed to be cache misses, no feasible schedules could be found for the first and the third taskset. For taskset 2, a feasible schedule taking 64310 clock cycles was found; this schedule is more than four times longer than any schedule generated taking cache effects into account. Comparing the results of methods C and I, we can see that the schedules calculated without considering extrinsic cache behavior, i.e. by assuming the empty cache as initial state, take up to 5% more time than the schedules taking the inter-task cache interference into account.
In our experiments, we have only considered relatively small task sets with low execution frequencies. Since the tasks contain many loops, the impact of extrinsic cache behavior is limited; for a set of frequently executed tasks of small size, the performance gain by considering the inter-task cache behavior will be significantly higher. An experimental evaluation will be included in the final version of this paper.

6 Conclusion

In this paper, we have presented a pre-runtime scheduling algorithm which takes the extrinsic cache behavior of the different tasks into account. Tasks are modeled as a sequence of segments in order to combine the flexibility of preemptive algorithms with the predictability of non-preemptive algorithms. The execution of segments of different tasks can be interleaved; the correctness of the resulting schedule is guaranteed by precedence constraints between the task segments. The intrinsic cache behavior is calculated by a static cache analysator generated by the program analysator generator PAG. The extrinsic cache effects are taken into account by analysing each task segment starting with the abstract cache state at the end of the preceding task segment in the schedule. This way, the switching costs can be incorporated into the scheduling process and the cache behavior of different task sequences can accurately be predicted.

7 Future Work

As already mentioned, we will extend our experiments by considering larger task sets and a larger range of cache architectures. At the moment, the implementation of the timing scheme of [LMW95] and its integration into our scheduler is under way. Also, we are considering the integration of data cache analysis into the scheduler, as well as a processor-pipeline analysis.
Further extensions concern the scheduling algorithm itself. Up to now, the extrinsic cache behavior is incorporated into the scheduling process, yet it cannot directly influence the scheduling decisions. The algorithm could benefit from a heuristic that takes the cache behavior of different task sequences into account, so that sequences with low switching costs would be preferred.

8 Acknowledgments

We would like to thank the Real-Time research group at the SNU for making publically available their test programs. Special thanks go to Christian Ferdinand and Florian Martin for their help in the integration of the PAG analyzer into our scheduler.

References

[AFMW96] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. *SAS'96 Static Analysis Symposium*, 1996.

[BFR71] P. Bratley, M. Florian, and P. Robillard. Scheduling with Earliest Start and Due Date Constraints. *Naval Res. Logist. Quart.*, 18:511–517, 1971.

[BMSO+96] J. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems. In *Proceedings of the 2nd Real-Time Technology and Applications Symposium*, pages 204–212. IEEE Computer Society Press, june 1996.

[BN94] S. Basumallick and K. Nilsen. Cache Issues in Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[CC77] P. Cousot and R. Cousot. A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[Fer97] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Universität des Saarlandes, September 1997.

[FMW97] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 37–46, June 1997.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, second edition, 1996.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The BCS Computer Journal*, 29(5):390–395, January 1986.

[LBJ+95] S.-S. Lim, Y. Bae, G. Jang, B.-D. Rhee, S. Min, C. Park, H. Shin, K. Park, S.-M. Moon, and C. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[LHYM+96] C.-G. Lee, J. Hahn, S. Y.-M., 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 Preemptive Scheduling. In *Proceedings of the 16th Real-Time Systems Symposium*, 1996.

[LMW95] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 298–307, December 1995.

[Mue94] F. Mueller. *Static Cache Simulation and its Applications.* PhD thesis, Florida State University, July 1994.

[MWH94] F. Mueller, D. Whalley, and M. Harmon. Predicting Instruction Cache Behavior. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

[PK89] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 2(1):159–176, September 1989.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time-Synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[TBW94] K. Tindell, A. Burns, and A. Wellings. An Extensible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *The Journal of Real-Time Systems*, 6(2):133–151, March 1994.

[TMLA97] S. Thesing, F. Martin, O. Lauer, and M. Alt. *PAG: User's Manual.* Universität des Saarlandes, 1997.

[XP90] J. Xu and D. L. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.