A Cache-aware Scheduling Algorithm for Embedded Systems

Gabriele Luculli Marco Di Natale
Università di Pisa Facoltà di Ingegneria dell'Informazione
Via Diotisalvi, 3 - 56100 Pisa - Italy
marco@pegasus.sssup.it luculli@cube.sssup.it

Abstract

This paper presents a methodology for scheduling real-time tasks in embedded systems where the task layout is known at design time and does not change at execution time (static systems) and where the cache miss costs are significant when compared to the normal execution time of the tasks. The scheduling model assumes a time-driven dispatching of the application tasks which are ordered in a pre-defined sequence. Building such a sequence in a way that is not only efficient but accounts for optimal cache sequencing is the aim of our method. The refinement of the schedule towards an optimal solution is done by simulated annealing techniques. The evaluation of the schedules is done by considering the effects of instruction caching when evaluating the computation time of the tasks.

1 Introduction

Although the main flow of the research is taking different directions, a significant portion of today's real-time systems are still static. By static we mean a system where the task parameters and among them their activation times, periods and deadlines, are entirely known beforehand.

Static systems are often found in embedded systems, where no explicit human interaction is foreseen and where the electronic components, CPUs and microcontrollers, act within a machinery to implement some or all of its functionality or simply to optimize its behavior. To give an example, most automotive control systems are still built as static systems and this should not surprise, as static schedules are more understandable and easier to build.

This does not mean that the research has solved all the issues on static scheduling, on the contrary, while some large scale or architectural solutions exist [1], there is still some confusion on how to handle the low-level aspects, like the scheduling or the timing analysis. These two aspects, are traditionally decoupled and studied separately. This is done only by convention and for sake of simplicity at the cost of a number of worst case assumptions.

A different approach is now emerging in the study of new design methodologies for embedded systems, where the major trend is the integrated study of the software policies and the hardware architectures supporting the computation, at the point that a new research area (co-design techniques [7]) has been created around these issues. The scheduling of the real-time computations is still out of the main research flow on co-design, but its importance is increasing and it is being acknowledged as an essential component.

The aim of this paper is to present a scheduling methodology that integrates scheduling and timing analysis taking into account the effects of caching, in a way that could be much more effective than the traditional methodology where the worst case effects of caching in the timing analysis are added to the worst case effects of the cache replenishment caused by a context switch (and both evaluations are far from being easy). In particular our analysis is aimed at reducing the pessimism of some of the worst case assumptions, and the repetitions caused by a two-phase view of the problem. The objective of this study is to have a better understanding of what could the effects of caching be not only on the task timing but also on the scheduling choices.

We assume a direct-mapped instruction cache of small dimensions 2-16 Kbytes with lines 8 to 32 bytes long, typical of most last-generation microcontrollers and microprocessor cores for embedded systems (for example the ARM family of microprocessor cores). Data caches can in principle be added to the model but at present time we do not handle them. Given the small dimensions of the cache the effects of intrinsic and extrinsic misses are particularly relevant. Extrinsic misses are caused by the interference of the preemption on the cache state and are unavoidable in multiprogrammed preemptive systems, unless a code partitioning policy [2] where cache sections can be pinned for each single task is used. Intrinsic misses are caused by the interference on the cache lines of the various process basic blocks and are unavoidable each time the dimensions of the task are greater than the cache dimensions (likely with small caches like those used in embedded systems), or when the task *Control Flow Graph* (CFG) does not allow an optimal allocation of the code (i.e. an allocation where the interference among the task's basic blocks is minimized). The use of recent allocation techniques aimed at the optimization of the performance by reducing the number of average misses [3] does not give absoute guarantees on the improvement of the worst case behavior, given that the reduction of the average cost comes together with the increase of the variance in the number of the misses. Code placement techniques can, however, improve the performance of such a system. We will make use of very simple assumptions on the placement of critical code sections (the clock interrupt handler and the task dispatcher) and we feel that the model presented here could (in theory) be extended to jointly evalute a sub-obtimal placement of the code of the tasks.

The effects of caching on the running time of real-time tasks in a multiprogrammed system are usually estimated in two separate stages. First comes a study on the cache aimed at the estimation of the worst case execution time of the tasks [4, 5], then there is another study on the cache refills at the context switches and their impact on the schedulability test [6].

We seek a real-time scheduling methodology for embedded systems where the two effects are jointly evaluated, as in reality, where the cache misses caused by context switches influence the number of misses in the internal execution of the tasks. Such an integrated study allows to have an estimate that is not excessively pessimistic and more natural then the time-consuming and difficult two-phase study on the worst-case impact of the cache refills.

One of the major features that characterize embedded applications is an extremely heterogeneous task set. Embedded systems contain extremely short tasks having an high activation rate, usually for polling the various sensors that exist in embedded systems, together with longer tasks, devoted to data processing and using the higher fraction of the CPU time at a much lower rate. This task configuration forces the scheduler to do a lot of preemption to let the short tasks have precedence upon the execution of the slow and large tasks for high level-control, data processing and strategy evaluation. The large number of context switches causes a lot of cache misses that drastically change the picture from the one that is expected when studying the execution time of the tasks one at a time. In other words, this scenario limits the usefulness of the conventional methods used to study separately the effects of the cache misses in the CFG of the tasks from the misses caused by the context switches and therefore the scheduling policy.

In summary, the main contributions of this paper are at least:

- the integrated analysis of the intrinsic and extrinsic cache misses and a less pessimistic evaluation of the worst case bounds; it is in fact possible to reduce the estimate on the largest number of cache misses that can happen in reality.
- The integration of a fine-grain scheduling procedure with the execution time estimation procedure. As a result, the scheduling choices can be influenced by the possible paths in the CFG of the tasks and the current state of the cache.
- A simple CFG based task model and a simple scheduling tool using it; a scheduling tool implementing the ideas contained in this paper is relatively simple and a possible implementation is suggested here.
- A very simple, accurate (includes the modeling of the instruction cache and processor pipelines) but potentially time consuming timing analysis of the code of the tasks.

The paper is structured as follows. Section 2 contains the analytical model of the tasks and the cache, the assumptions on the scheduling and the application environment. It also contains some approximations that are necessary in order to perform the analysis with a reasonable occupation of space and time. Section 3 contains a description of the scheduling procedure, its limitations and advantages. Section 4 contains an example of the scheduling methodology applied to a small embedded application. Finally, section 5 contains a description of a proposed scheduling tool that makes use of the methodology proposed in this paper and a discussion on the usefulness and the applicability of the method; the bibliography ends the paper.

2 Models and Assumptions

This section contains the definition of the scheduling problem, the description of the task model and the cache model. The modelization we propose includes all the parameters that are necessary to the joint evaluation of the cache effects and the process timing in building the schedule. As shown in one of the following sections the task model needs a further refinement from the classical Liu and Layland's couple (C_i, T_i), representing a task by means of its execution time and period.

An extended task model is not the only thing that we need for our analysis. We aim at a representation that includes the effects of caching and a timing analysis, therefore a cache and processor model is needed, together with the description of the functions that operate on both the task and the cache model to extract the information that is necessary to our study.

But the first thing to be described is the scheduling model that we propose and the scheduling problem that we need to solve.

2.1 The Scheduling Problem

We choose to use a modified time driven scheduler [1], that is a scheduler where the decisions are taken either when a task terminates or when a timer interrupts after a given amount of time. This choice is motivated by two reasons: first, time driven systems are popular enough to make the model acceptable for practical uses, second and most important, the essence of the method is to understand where in the task code is more effective to put the preemption points.

By slicing time we have a static or off-line correspondence (even if one to many) between the possible scheduling points and the task code in the CFG, and it is possible to exploit this correspondence to build an efficient schedule. A different and perhaps more intuitive approach would be to set the possible preemption points inside the task structure, but if the preemption points were bound to the code of the tasks, the scheduling choices would depend on the exact execution path of the tasks which is known only at run-time (dynamic scheduling) and it would be very difficult to give optimality rules or guarantees before run-time.

Our problem is to schedule a task set $\tau = \{\tau_1, \tau_2, \tau_3, \dots \tau_N\}$ where each task τ_i is known a priori and is defined by an annotated control flow graph (CFG_i), a period T_i and a deadline D_i not necessarily coincident with the period. The tasks are placed in memory at fixed locations (i.e. task placement is not part of our analysis). The problem can also include the sharing of resources among tasks and precedence constraints among them. We skip these additional constraints by now, but we will see how they can be easily included in the model in the following. Our objective is to build an optimal task ordering in a time interval equal to the least common multiple of the periods of the tasks or *scheduling cycle*. This ordering consists in the assignment of a time slot to each task for its execution and it can be stored in a table to be used at run-time by a simple dispatcher, triggered by the internal clock.

We can initially assume that all the time slots in the *scheduling cycle* have the same length. This makes the scheduling model simple although not very efficient. Our time driven scheduler is based on the execution of a dispatching routine, called by the driver of the clock interrupt. The dispatcher reads in a table which task is to be executed in the next slot. In the simplest fixed slot model, the clock interrupt driver and the dispatching routine execute on every slot, even when a context switch is not needed (see Figure 1). This approach unnecessarily wastes precious CPU time.

A refinement of this strategy is to have time slots of different length (although multiple of some base size) so that each slot boundary corresponds to a context switch (Figure 2). This approach avoids unnecessary runs of the clock driver and the dispatching routine at the price of a slightly more complex interrupt driver and dispatching routine. In section 3 we will see how to generate time slots of different length.

In the general case, the dispatching table consists of

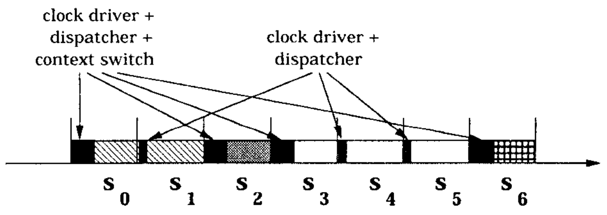

Figure 1: Execution pattern with slots of fixed size

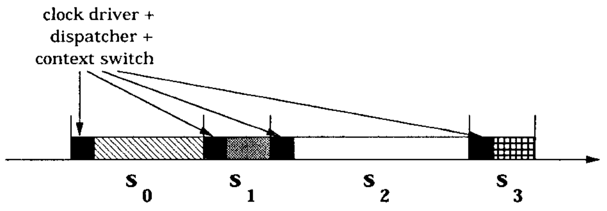

Figure 2: Execution pattern with slots of variable size

a list of context switch descriptors. Each context switch descriptor is a pair of the form

$$CS_j = (t_j, \tau_j)$$

containing the task that is to run on the cpu as a result of the context switch and the time when the context switch must take place, expressed as relative to the beginning of the *scheduling cycle* or (even better) as relative to the previous context switch. In the general case the routine for handling the clock interrupt has the form of Figure 3, where each line can easily correspond to a short number of assembly instructions.

In order to enhance both the predictability and the performance of the system, we assume the driver of the timer interrupt and the dispatcher routine are pinned in cache, using blocks that are not part of our analysis.

The aim of any real-time scheduling algorithm is to produce a *feasible schedule* that is a schedule where all tasks complete before their deadline. We make no exception to this rule, but this concept of schedulability doesn't fit too well with the definition of an optimality metric to choose among different schedules as it only returns a binary value. In particular, as we will see, it doesn't fit at all the needs of a simulated annealing scheduling algorithm.

The metric that we propose to evaluate the quality of the schedules is instead a laxity metric. That is, the value of each schedule is the minimum laxity of its tasks, where the laxity $l(\tau_i)$ of a task τ_i is the difference between its deadline and its completion time. The laxity not only gives a measure of the schedulability (i.e. all schedules with minimum laxity greater than 0 are feasible) but also gives a measure of the robustness of the schedule. What we mean by optimal schedule, then, is a schedule that maximizes the minimum laxity of the task instances.

```
clk_IH()
{
  dispatch_index = (dispatch_index+1)%cycle_length;        /* incrementing the dispatch index  */
  clear_interrupt();                                        /* clear the interr. flags */
  reload_timer(CSD[dispatch_index].newtime);                /* set for next slot */
  enable_interrupt();                                       /* enable interrupts */
  dispatch(CSD[dispatch_index].newtask)                     /* go to new task */
}
```

Figure 3: Scheme of the clock interrupt driver

2.2 The Task Model

We need a description of the tasks that allows the modeling of the cache and where not only the allocation of the task code is included, but also the order in which the cache lines are possibly filled during the execution of the task.

This paper assumes a task model based on annotated Control Flow Graphs (CFGs). The annotations are both on the basic-blocks and on some edges. By basic block $B_k^{(i)}$ of a task τ_i we mean a sequence of executable code which is executed sequentially (i.e. does not contain any external or conditional branches).

In particular, each basic block $B_k^{(i)}$ is associated a memory interval $[Start_k^{(i)}, End_k^{(i)}]$, spanning a number $N_k^{(i)}$ of bytes and the execution time corresponding to an execution with no cache misses $C_k^{(i)}$. The only annotation on the edges is on the back edges of loops and corresponds to the loop bounds necessary to have a bound on the execution time of the task.

With this definition, each task model ends up being composed of three elements: an annotated CFG, an activation period and the task deadline $\tau_i(CFG_i, T_i, D_i)$. Please note that the proposed model is a reasonable extension of the classical pair $\tau_i(C_i, T_i \boxminus D_i)$, where C_i is the worst-case execution time. This extension is necessary to include the cache effects in the scheduling analysis and at the same time does not exclude other possible extensions. This model fits the infeasible path analysis [8], the modeling of other hardware architectural features such as pipelining and pre-fetch buffering, and possible future extensions. In particular, the effect of pipelining within each basic block can already be accounted for and added in the computation of the $C_k^{(i)}$.

2.3 The Process Model

The process $PC^{(i)}(t)$ associated to the task τ_i, is the execution state of τ_i at time t. The process is function of the schedule and is represented by two elements: the basic block in which each task is currently in execution at t, and by an offset within that basic block.

$$PC^{(i)}(t) = (B_{k(t)}^{(i)}, Off^{(i)}(t))$$

Unfortunately, the worst case execution path is unknown beforehand, as it depends on multiple factors such as the input data, or the exact cache state at the context switch times and therefore on the scheduling decisions made up to the given point and the CFGs of the concurrent tasks.

If we cannot exclude apriori any path in the CFGs of the tasks, then we have to consider all the possible combinations of them, their effect on the cache and on the schedule decisions taken at each slot.

This uncertainty on the actual execution path can be modeled by extending the concept of process $PC^{(i)}(t)$ of the task τ_i at time t. The process is represented not only by the single state $(B_{k(t)}^{(i)}, Off^{(i)}(t))$ but by the set of all the possible pairs corresponding to all the possible combinations of execution paths.

In other words, if $P^{(i)}(t)$ is the set of all the possible paths for the CFG_i in the time interval $[0, t]$, we have

$$PC^{(i)}(t) = \{(B_{k(t)}^{(i)}, Off^{(i)}(t))_j : j \in P^{(i)}(t)\}$$

In other words, we have to consider all the possible paths in the CFG. In the worst case this means that the number of elements in the process set is exponential. In reality, it is reasonable to hope this number will not be too large, because in most real-world embedded applications most of the tasks are short and the branching factor is limited. In Figure 4 at time t_0 the process associated to the task τ_1 is $PC^{(1)}(t_0) = (B_1^{(1)}, 0)$, at time t_1 the process has two items, corresponding to the two possible execution paths, therefore $PC^{(1)}(t_1) = \{(B_3^{(1)}, x)(B_4^{(1)}, y)\}$

2.4 The Cache Model

Our cache model is for instruction-only, direct-mapped caches. A data cache modeling could in principle be added to the study, given that in most embedded programs a lot of variables are global and therefore statically allocated and that some studies are already showing how to study the performance of data caches; a more realistic model should deal also with a set-associative cache. the study of such a model is currently under development and will hopefully be presented in the future.

The cache is modeled by an abstract representation of the cache state. Two functions will be used to change the abstract state or extract information from it. By abstract state we mean an approximation of the real one,

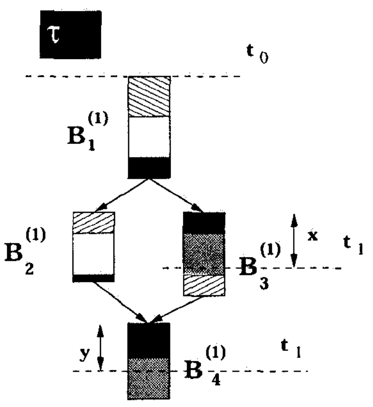

Figure 4: Process basic blocks and the state of the process in t_1

aimed at obtaining the information needed to pursue our timing and scheduling analysis.

The goal of the analysis is, given a segment on the CFG and an abstract cache state, to determine the number of misses generated in the worst case by the execution of the CFG segment and the cache abstract state at the end. We need to define two basic abstract functions. The first $Next(B^{(i)}_{k.l}, \Pi(t))$ function of a program line and of the current cache state, returning the new cache state. The second $Miss(B^{(i)}_{k.l}, \Pi(t))$ function of a program line and of the current cache state, returning 1 or 0 according to the occurrence or not of a miss. The miss number and the final cache state produced by a segment on the CFG, is given by applying the functions *Next* and *Miss* to the basic block's sequence of the segment.

Definition: Every basic block $B^{(i)}_k$ is composed of a given number of program lines $(B^{(i)}_{k.1}, B^{(i)}_{k.2}, B^{(i)}_{k.3}, \ldots)$. Each program line $B^{(i)}_{k.l}$ corresponds to one and only one cache line $CacheLine(B^{(i)}_{k.l})$.

Definition: At any time the abstract state of the cache $\Pi(t)$ is the union of the abstract state of its L cache lines $\pi_j(t)$:

$$\Pi(t) = \bigcup_{j=0}^{L-1} \pi_j(t)$$

Definition: Given a task set $\tau = \{\tau_1, \tau_2, \tau_3, \ldots \tau_N\}$, and a partial schedule $\sigma(t)$ on $[0, t]$. The abstract state of the cache line $\pi_j(t)$ is the set $\{B^{(i)}_{k.l}\}$ of all the program lines that could possibly reside in it at time t, where $B^{(i)}_{k.l} \epsilon \{B^{(m)}_{p.q} : CacheLine(B^{(m)}_{p.q}) = j, \ m \epsilon \tau\}$.

Definition: Given a program line $B^{(i)}_{k.l}$ and a cache state $\Pi(t)$, the miss abstract function *Miss* is defined as follows:

$$Miss(B^{(i)}_{k.l}, \Pi(t)) = \begin{cases} 0 & \text{if } B^{(i)}_{k.l} \in \pi_j(t), \\ & j = CacheLine(B^{(i)}_{k.l}), \\ & TaskIn(\pi_j(t)) = \{i\} \\ 1 & \text{otherwise} \end{cases}$$

where *TaskIn* returns all the indexes of the tasks which actually are in a cache line.

Definition: Given a program line $B^{(i)}_{k.l}$ and a cache state $\Pi(t)$, the next state function *Next* is defined as follows:

$$Next(B^{(i)}_{k.l}, \Pi(t)) = \bigcup_{j=0}^{L-1} next(B^{(i)}_{k.l}, \pi_j(t))$$

$$next(B^{(i)}_{k.l}, \pi_j(t)) = \begin{cases} \pi_j(t) & \text{if } j \neq CacheLine(B^{(i)}_{k.l}) \\ \{B^{(i)}_{k.l}\} \cup \Delta & \text{otherwise} \end{cases}$$

where Δ is the empty set if $B^{(i)}_{k.l}$ is the only program line that maps into the cache line, otherwise Δ is the set of the other program lines of τ_i which are loaded into the same cache line at the same time.

An abstract state of a cache with 4 lines and its evolution as a result of the execution of a task segment is represented in Figure 5

3 The Scheduling Methodology

The scheduling methodology can be any one of the search methods used for solving NP-complete static scheduling problems. Search methods based on evaluation heuristics are acceptable, as well as other methods based on probabilistic analysis or simulated annealing. The difference between the two methods is the way that is used to reach a solution close to the optimal one according to the definition of optimality given in subsection 2.1.

Conventional search methods, like those based on pruning [12] or IDA^*/branch and bound, try to build the optimal solution incrementally, by building sub-schedules and deciding which task (or task slice) must follow next, based on the value of the task in the given scheduling position (pruning) or on the distance of the incremental schedule from the optimal solution (branch and bound). The choices made in searching for the optimal solution can eventually be undone by backtracking if the procedure comes to a dead end, although this should be avoided as much as possible.

On the other side, simulated annealing techniques consist in trying complete scheduling solutions, no matter how good or efficient, and moving from a solution to neighboring solutions by means of a probabilistic operator that is based on the optimality metric. The newly found solutions are evaluated and conditionally accepted on the basis of their cost.

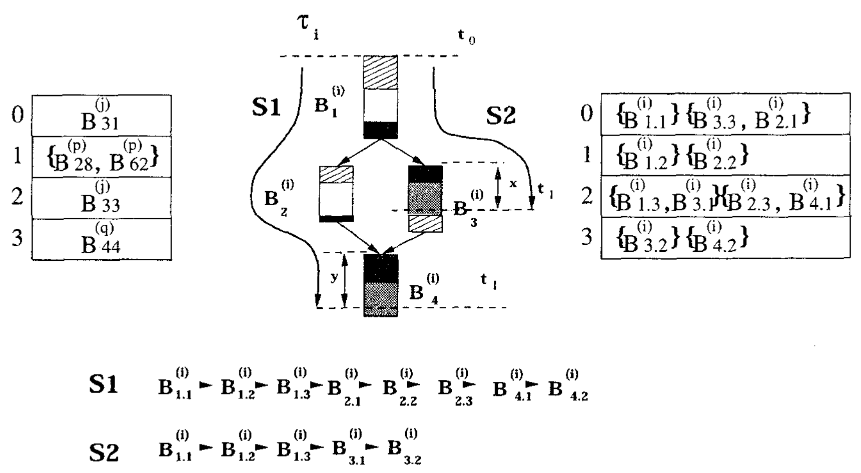
τ_i
t_0
S1
S2
$B_1^{(i)}$
$B_2^{(i)}$
$B_3^{(i)}$
$B_4^{(i)}$
x
y
t_1
0 $B_{31}^{(j)}$
1 $\{B_{28}^{(p)}, B_{62}^{(p)}\}$
2 $B_{33}^{(j)}$
3 $B_{44}^{(q)}$
0 $\{B_{1.1}^{(i)}\}\{B_{3.3}^{(i)}, B_{2.1}^{(i)}\}$
1 $\{B_{1.2}^{(i)}\}\{B_{2.2}^{(i)}\}$
2 $\{B_{1.3}^{(i)}, B_{3.1}^{(i)}\}\{B_{2.3}^{(i)}, B_{4.1}^{(i)}\}$
3 $\{B_{3.2}^{(i)}\}\{B_{4.2}^{(i)}\}$
S1 $B_{1.1}^{(i)} \to B_{1.2}^{(i)} \to B_{1.3}^{(i)} \to B_{2.1}^{(i)} \to B_{2.2}^{(i)} \to B_{2.3}^{(i)} \to B_{4.1}^{(i)} \to B_{4.2}^{(i)}$
S2 $B_{1.1}^{(i)} \to B_{1.2}^{(i)} \to B_{1.3}^{(i)} \to B_{3.1}^{(i)} \to B_{3.2}^{(i)}$

Figure 5: The abstract state of the cache

Local search techniques whould accept only solutions having an higher value (or lower cost), at the risk of moving towards local minima and remain stuck there. Simulated annealing tries to escape local minima by accepting higher cost solutions with a probability given by an exponentially decreasing function of the energy difference like

$$Prob(\Delta E) \sim e^{\frac{-\Delta E}{kT}} \qquad (1)$$

The conditional acceptance of higher cost solutions is gradually reduced, by decreasing the parameter T (acting like a temperature). The situation is similar to the one of a liquid that cools and solidifies. This similitude explains the name of the method.

By modeling the series of transitions as a Markov chain, it was proven that when the temperature T converges to 0, the stationary distribution has probability one for the set of the optimal (minimum cost) solutions, 0 for the others. This means that the method is eventually able (actually after an undetermined amount of time) to reach the optimal solution(s).

Going back to the comparison between classical search methods and simulated annealing, the advantage of the classical search methods is their potential for a much higher speed, given that a good pruning strategy or evaluation function is found. The disadvantages are the worst case exponential running time and the difficulty (sometimes) of finding good evaluation functions. The disadvantage of simulated annealing techniques is a large execution time, in most cases of the order of the third or fourth power of the number of schedulable units, although always polynomial. The advantages are its simplicity, flexibility and the fact that the method consists in the refinement of sub-optimal solutions, allowing the user to stop the process at will, whenever a solution that is good enough for his purposes is found.

Now, let's go back to the scheduling problem and the particular solution that we propose for it.

We are interested in building a static schedule, that is a layout of all the periodic task instances in the *scheduling cycle* or least common multiple of the periods. From here on, by task we mean an instance of a periodic task, having its start time r_i, and its deadline d_i. Therefore, the k-th instance of task τ_i will be activated and has its start time at $r_i = (k-1)T_i$ and must terminate before its deadline at $d_i = (k-1)T_i + D_i$. Suppose all task instances are numbered from 1 to N, according to any order.

If we lay out the time slices in the least common multiple of the periods of the tasks, we have a sequence of slots like in Figure 1; each slot can be assigned to one or more tasks (in case the first task ends in the middle of the slot).

```
anneal(num_slots, task_list, temperature, coolrate)
{
  assign_slots_to_task_lists();
  build_schedule();
  value = estimate_schedule_value();
  for (j=1; j <= MAXCHAINS && temperature > MINTEMP; j++) {
      nsucc = 0;
      for (k=1; k <= MAXTRY; k++) {
                assign_slots_to_task_lists();
                build_schedule();
                newvalue = estimate_schedule_value();
                deltavalue = value - newvalue;
                valid = ischange(deltavalue, temperature);
                if (valid) {
                        nsucc++;
                        value = newvalue;
                        confirm_change();
                } else
                        undo_change();
                if (nsucc >= MAXCHANGE)
                        break;
        }
        temperature *= coolrate;
        if (nsucc == 0) return;
        }
}
```

Figure 6: Core of the simulated annealing scheduling algorithm

We choose to obtain an efficient assignment of the slots to the tasks (a schedule close to optimality) by using a simulated annealing algorithm of the kind in Figure

6. The reason of the choice of a simulated annealing solution is the difficulty of this particular scheduling problem, surely much harder than conventional scheduling. We feel that, in our case, finding an efficient feasible evaluation function for a branch and bound heuristic might prove to be extremely hard (if possible at all).

The simulated annealing algorithm in Figure 6 consists in two nested loops. The outer loop corresponds to the series of markov chains that are generated for decreasing values of the temperature parameter. The inner loop, embedding the core functions: the generation of the new solutions; their evaluation and their conditional acceptance, generates the single elements of each markov chain.

In particular, the generation and evaluation of the new schedules are the most important parts while the layout of the simulated annealing routine is pretty general [11] and can be applied to a variety of other scheduling problems [10]. The steps necessary to the construction and evaluation of a new schedule are the following:

- the algorithm performs a partial random assignment of the time slots to the tasks in a way that is consistent with the task requirements. This assignment cannot be unique, as we do not know beforehand the exact computation time of the tasks. The latter in fact depends, even in the worst case, by the amount of cache misses.
- Starting from the first slot and following the time sequence, the slot assignment will be translated in the actual timing evolution of the tasks, including the interactions in cache caused by the task interleaving in the schedule, the intrinsic misses and the time necessary to run the dispatcher and the driver of the clock interrupt.
- The evaluation of the exact task timing allows the ongoing choice of one possible assignment of the slots to the tasks among those proposed in the first phase; if two (or more) consecutive slots are assigned to the same task, the slots are coalesced into one and the new slot size is saved in the corresponding context switch descriptor. The time for running the driver of the clock interrupt and the dispatcher routine is assigned only at the beginning of the first slot.
- At the end of the process, not only the randomly generated schedule will be exactly characterized, but the whole schedule will be evaluated on the basis of the minimum laxity of the component tasks.

Now, let's examine how the single phases can be implemented.

The first step of the simulated annealing algorithm is the assignment of the slots to the tasks; this assignment cannot be deterministic but should rather consist of a

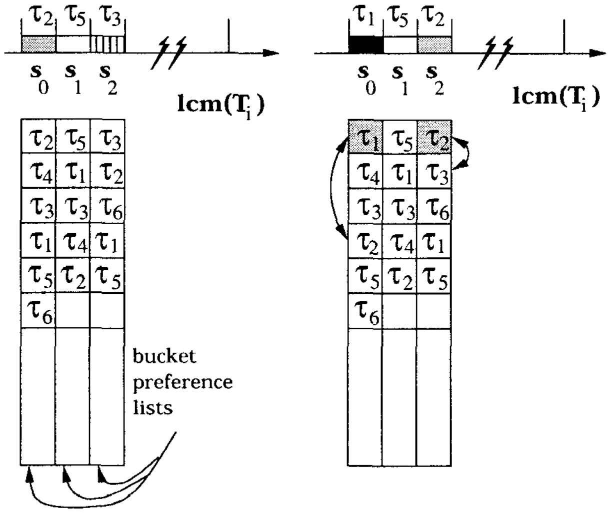

Figure 7: Bucket preference lists and the generation of a new schedule through the modification of the lists

rule to guide the choice of the task that runs in the slot. Therefore, the algorithm assigns a bucket to each slot, containing a preference list. The list contains all the tasks that are *schedulable* in the slot, ordered by priority. A task τ_i is *schedulable* in the slot $s_j = [t_{j-1}, t_j]$ if $[r_i, d_i] \cap [t_{j-1}, t_j] \neq 0$.

A new schedule is constructed by means of the following transition operator TR. The simulated annealing routine builds new schedules by changing the task order in the bucket lists. The TR operator randomly chooses a number of lists that are to be changed (minimum 1, maximum N). For each list selected for a change, the operator switches the first task with a randomly chosen task that follows in the list. For example, in the right side of Figure 7 the first and third list have been chosen. In the first list the first task exchanged positions with the fourth task. In the third bucket the first task has been exchanged with the second in list. this is an easy and fast way of generating new schedules that are consistent with the task requirements.

The second step is done by sequentially examining the slots and assigning them the first *available* task in the bucket list, and updating the process state of the assigned task and the cache state as a consequence of the assignment. By *available* task in the list we mean a task which is *active* (i.e. its activation time has elapsed) and not *late* (i.e. its deadline has not expired) and is not terminated yet (i.e. it has not been assigned execution time up to its maximum completion time or until the end of its CFG). This concept of *availability* can be easily extended to handle task configurations where resources are shared among tasks in exclusive mode or where there are precedence constraints among tasks. In particular, a task is not *available* if requesting a resource that is currently in use by some other task, and is not available until all its predecessors have completed their execution. Please note that this formulation allows for the definition of precedence constraints not only among tasks, but also among their component basic blocks. All tasks that are not *available* are simply ignored in the bucket list.

Once the time slot has been assigned to the first *available* task, we evaluate the time evolution of the task (its process) and the cache state. This is done with the following sequence of actions. First the task is compared to the task holding the previous slot. If they are the same task, the new slot is joined with the previous one, otherwise the algorithm reserves some time for the execution of the clock driver and the dispatcher. Then, for each item in the process of the task, the algorithm considers the next process line and continues processing the process lines until the end of the time slot is reached.

While processing the process lines, new process items can be generated (for example because of forks in the CFG) or deleted, in case the corresponding execution path terminates. Processing each process lines means evaluating the $Next$ and $Miss$ functions defined in subsection 2.4. The $Miss$ function determines if a cache miss is possible before the execution of the line. If so, time is reserved for the miss before the time necessary to execute the line itself. The $Next$ function evaluates the state of the cache at the end of the line. As a result, at the end of the second phase and when the last slot has been examined, the slots are deterministically assigned to the tasks. This operation concludes the third step as well.

At this point, it remains to perform the evaluation of the metric function. When the last slot has been assigned and all the termination times of the tasks have been computed, at the worst case exiting points from the CFG, the schedule can be examined and the termination time of the tasks compared to their deadlines, evaluating the minimum laxity in the system.

4 Example: A Simple Filtering Application

To show the characteristics of the scheduling methodology we developed a small example based on the ARM610 microcontroller core, typically used in many embedded applications. Our application consists of a set of tasks belonging to a filtering module in a real application. In the following we will give the description of the task set, the hardware characteristics and the partial schedule $\sigma = \{\tau_1, \tau_2, \tau_3, \tau_2\}$ shown in detail in Figure 9.

The task set $\{\tau_1, \tau_2, \tau_3\}$ consists of three tasks implementing a filtering module. A stream of signal samples arrives as an input and the stream must be filtered in an optimal way according to the dynamic characteristics of the stream itself. The application includes two types of filters, both implemented as a single FIR acting with different coefficients. The signal characteristics allow a software filtering, thus saving on the implementation costs, but the need for an adaptive optimal filtering requires the use of several hardware buffers. The task τ_2 is an assembly procedure highly optimized for the ARM family of controllers. Its code has been obtained from the ARM applications library and it implements a FIR filter with a shift-accumulate structure with 120 taps.

The input samples are read from an hardware memory mapped buffer and are subsequently stored in another buffer after the computations. The task τ_1 is a procedure written in C that performs a quick analysis on a small section of the input stream, selects the optimal filtering between the two that are available and (upon availability of an hardware buffer), it starts τ_2 with the appropriate parameters.

The task τ_3 is a short assembly procedure that polls τ_2 to check for its termination, and sets some hardware flags to signal τ_1 the availability of hardware buffers and the end of the filtering process to the other application tasks (or hardware modules). The time characteristics of the tasks can be derived from the requirement of not losing input samples, hence a deadline for τ_2, and from the utilization bounds for the hardware buffers, hence the activation rate of τ_3 and the characteristics of τ_1. A static schedule fits well these application requirements, since all the timing requirements are known at design time and do not change at run-time.

The hardware architecture consists of a ARM610 microcontroller core, a 32 bit, 33 MHz clock RISC processor with 1kbyte of ROM memory containing the program code and a direct-mapped instruction cache of 128 bytes with a 16 bytes line size (4 instructions per line). Other parameters of the scheduler are a context switch overhead of 50 cycles, a cache miss latency of 20 cycles and a base time slot of 400 cycles.

All time and size parameters have been scaled of at least a 10 factor from what could have been the real-world values. This is done only for the purpose of clarity, otherwise all figures and tables would not fit in a page and our discussion would be limited to a meaningless single time slot. This should not limit the validity of the example and the usefulness of the method which is clearly not dependent from the exact values of these parameters. The dimensions of the cache have been shortened consequently in order to highlight the effect of misses. In a real application the dimension of the cache is higher, but also the time slots are larger and the overhead (caused by context switches and cache misses) longer. All these effects should somehow compensate.

In Figure 8 the CFGs of the three tasks are shown, as obtained from the assembly listings, the annotations on the arcs corresponding to the loop bounds, the memory intervals used by each basic block $[Start_k, End_k]$ and their size in bytes, and the execution time in cycles (en-

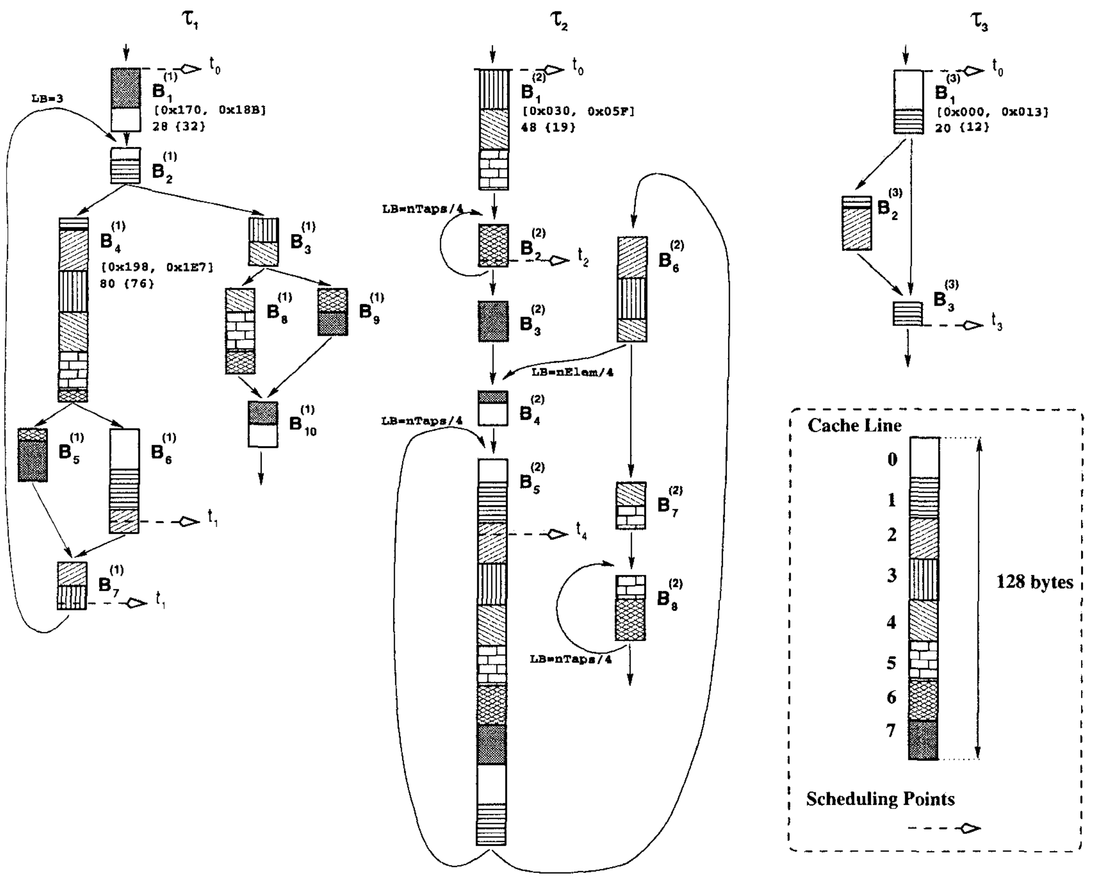

Figure 8: The annotated CFG of the application tasks in the example

closed in braces) for each basic block $B_k^{(i)}$. In Figure 8 the graph annotations are shown only for the first basic block. This is done only to keep the figure readable.

All the values shown in the figure and used for this example are the real values obtained from the analysis of the real code. In particular, the size and the execution time of each basic block have been obtained with a tool available in the ARM Development Kit, while the memory addresses have been obtained from an allocation of the code of the tasks which is consistent with the allocation rules of the ARM linker (first the assembly modules, then the C modules). Some of the loop bound are expressed as parameters. In the example ntaps is 120. Each basic block has been divided in program lines, assigning each cache line a different color.

In order to explain the proposed cache-aware scheduling method, we show in Figure 9 and describe in the following paragraphs the evaluation phase for the partial schedule $\tau_1(t_0), \tau_2(t_1), \tau_3(t_2), \tau_2(t_3)$. In particular, we show the possible schedules evaluated on each slot with the relative set of possible paths, and the cache state at the end of each slot. With reference to the previous sections it can be seen that

- In the time slot $[t_0, t_1]$ τ_1 is executed starting with the cache empty. There are four possible paths in the CFG of the task and inside the time slot; two of them terminate before the end of the slot and do not correspond to the worst case execution of the task. In the future, when τ_1 is scheduled again for execution, only the two remaining paths need to be considered. The state of τ_1 at the end of the slot is $PC^{(1)}(t_1) = (B_6^{(1)}, offset), (B_7^{(1)}, offset)$. The state of the cache at the end of the slot (actually at the end of each slot) is shown in the bottom half of the picture.

- The portions of the cache enclosed in a dashed line are those modified in the current slot. In the first slot τ_1 accesses all the cache, generating first refill misses and later intrinsic misses caused by the blocks $B_{5.2}^{(1)}, B_6^{(1)}, B_7^{(1)}$. Both type of misses are detected by the $Miss$ abstract function and considered in the generation of the four possible execution lines inside the slot (grayed portions of the schedule).

- The task τ_2 is executed in the second slot. There is only one possible path and a single internal schedule. In particular, the loop in $B_2^{(2)}$ (with a loop

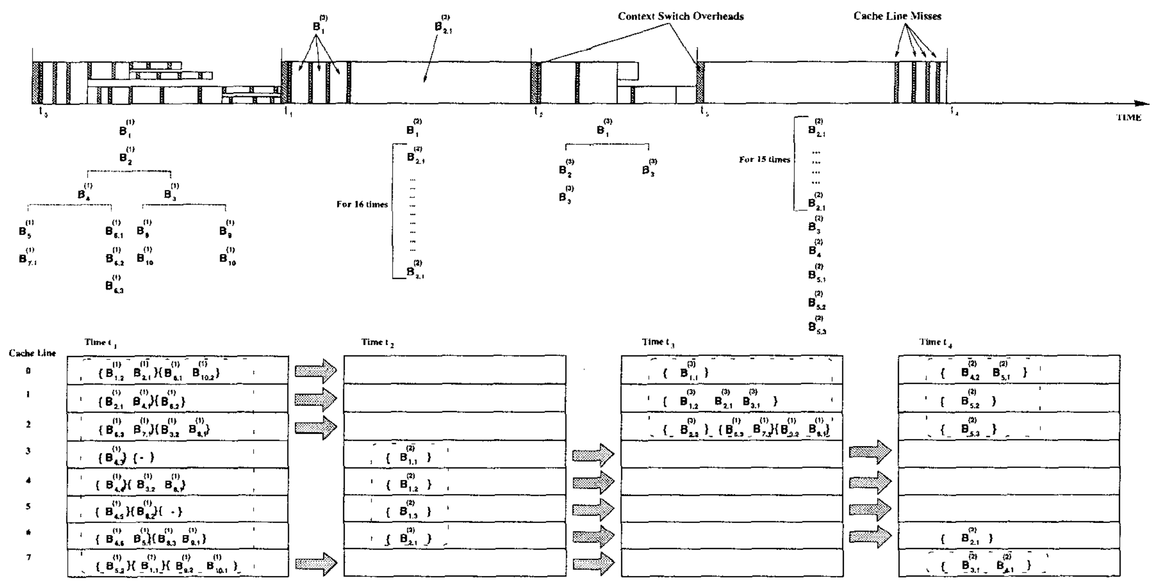

Figure 9: A partial schedule and the corresponding cache states

bound of 30 cycles) is executed fifteen times and a half, first giving a miss and then 15 consecutive hits, all correctly estimated by the *Miss* function.

- The task τ_3 is executed in the third slot. Since τ_3 ends before the end of the slot, the latter is shortened. Please note the abstract state of the third cache line at the end of this slot; it contains a program line belonging to τ_3 and program lines belonging to τ_1 that are present in cache since t_1 and could remain there even after t_3, depending on the following schedule (in this case they are removed from the following execution of τ_2).

- In the third slot, both the possible execution paths influence the abstract state of the cache in t_3, even though onyl one corresponds to the worst case timing. As both paths are possible, we cannot exclude the shortest one (and its effects on the cache).

- The task τ_2 is executed in the fourth slot. τ_2 ends its loop contained in $B_2^{(2)}$ without any miss. The *Miss* function correctly predicts 15 hits, differently from the classical worst-case methods which would add an initial miss.

In conclusion, the estimate on the possible hits and misses are pretty close to the reality, the only source of uncertainty in this example and in general is caused by the equivalence of all the possible paths in the CFG. Our future work includes the study of how this uncertainty could be drastically reduced by adding smarter annotations to the CFG of the tasks. For example, in our example, if we could add the constraint that the loop in task τ_1 is always executed three times (as it is in reality), two of the paths in the first slice would disappear, the abstract state of the cache in t_1 would be greatly simplified, and a number of non-existing worst-case states (and therefore reported misses) would be removed.

5 Conclusion and discussion

We are currently building a scheduling tool based on simulated annealing and following the description given in this paper. The aim is to try real-world examples (like the routines that can be found in the ARM Applications Library) to see what are the potential benefits of the method.

The tool takes in input the description of the application tasks (CFGs, periods, deadlines) and the description of the cache (overall size, line size) and gives out the schedule expressed by the list of the context switch descriptors. Each iteration in the scheduling tool consists in the creation of a new schedule, its evaluation and its conditional acceptance, according to the general algorithm in Figure 6.

The subject of this paper is a methodology for building a static time driven schedule of real-time tasks for embedded systems. The scope of the scheduler is to assign each task a time slot of variable size in such a way that the number of cache misses is significantly reduced when compared to a normal real-time schedule which does not account for the effects of caching.

The scheduler performs not only an efficient assignment of the CPU to the tasks, but also a timing analysis of the task code itself, allowing a more precise estimate of the effects of caching (number of misses). This can give advantages in comparison to the classical methods of evaluating of the worst case number of misses in the internal execution of the tasks added to the worst case number of misses in context switches.

In a few words, the method is expected to work in the following scenario: assume a program trace like the one in Figure 10 which is taken from a real–world example [9]. The figure shows how a normal execution trace presents points where the number of cache misses suddenly increases, followed by points where the number of misses is relatively low. Suppose a context switch is to be placed in the vicinity of one of those peaks. Two scenarios can happen.

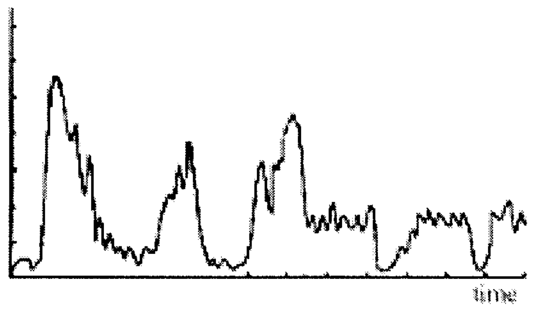

Figure 10: Cache misses in a trace of a real program

In the first, a context switch happens right at the peak, this means that the task caused a large number of misses only to have the CPU taken away and given to another task. The newly arrived task will cause another large number of misses, this time to load its program into the cache. When the CPU returns to the first program, it will essentially reload all its blocks into the cache, thus causing a large amount of misses for the third time.

In the second possible scenario, the CPU is given to the second task right before the peak in the number of the cache misses. This means that we can expect two sets of misses corresponding to the context switches, but when the first task resumes the control, the set of cache misses that follow the context switch will be exactly those in the peak of the trace graph. This results in two large sequences of cache miss rather than three. The scope of this work is exactly to pick the context switch points at the right times in the execution of a task program.

The research line proposed in this paper is not easy, the method proposed here could well prove to be etremely sensitive or even to fail in most cases. To be fair, it will probably take time before its impact on real-world applications could be proved or even seen. More realistic models are needed (set associative caches and probably modeling the data cache) and a large study on real world hardware architectures and programs. On the other hand we think that the problem is very interesting and a solution of the type proposed here could have a large impact on the traditional way of designing real-time embedded systems and studying the execution time of tasks and the effects of caching.

References

[1] H. Kopetz et al. *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach* IEEE Micro, 9(1):25-40, Feb. 1989.

[2] D.B. Kirk. *Smart (strategic memory allocation for real-time) cache design.* In Proceedings of the 10th Real–Time Systems Symposium, pages 229–237, December 1989.

[3] H. Tomiyama and H. Yasuura. *Optimal code placement of embedded software for instruction caches.* In Proceedings of ED&TC96, pages 96–101, 1996.

[4] Y.-T. S. Li, S. Malik, and A. Wolfe. *Performance estimation of embedded software with instruction cache modeling.* In International Conference on Computer Aided Design, pages 380–387, Los Alamitos, Ca., USA, November 1995.

[5] R. Arnold, F. Mueller, D.B. Whalley, and M. Harmon. *Bounding Worst-Case Instruction Cache Performance.* In Proceedings of the 15th Real–Time Systems Symposium, pages 172–181, December 1994.

[6] C.G. Lee, J. Hahn, Y.M. Seo et. al. *Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling.* In Proceedings of the 17th Real–Time Systems Symposium. December 1996.

[7] G. De Micheli. *Hardware/Software Co-design: Application Domains and Design Technology.* In Proceedings of NATO Advanced Study Institute, Tremezzo(I), June 1995.

[8] C.Y. Park. *Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths.* Journal of Real Time Systems 5(1), March 1993.

[9] J. Simonson and J. H. Patel. *Use of Preferred Preemption Points in Cache-Based Real-Time Systems.* Proceedings of IEEE IPDS, Erlangen, Germany, April 24-26, 1995

[10] M. Di Natale and J. A. Stankovic. *Applicability of Simulated Annealing Methods to Real-Time Scheduling and Jitter Control* . Real Time Systems Symposium, Pisa, December 1995.

[11] E. Aarts, P. Van Laarhoven. *A New Polynomial Time Cooling Schedule.* Proc. IEEE Conf. on Computer Aided Design, Santa Clara, 206-208, 1985.

[12] K. Ramamritham, J. Stankovic and P. Shiah. *Efficient Scheduling Algorithms For Real-Time Multiprocessor Systems.* IEEE Transactions on Parallel and Distributed Computing, Vol. 1, No. 2, pp. 184-194, April 1990.