Journal of Systems Architecture xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems ☆

Luis C. Aparicio [a,d], Juan Segarra [a,c,d,*], Clemente Rodríguez [b,d], Víctor Viñals [a,c,d]

[a] DIIS, Universidad de Zaragoza, 50018 Zaragoza, Spain
[b] DATC, Universidad del País Vasco, 20018 San Sebastián, Spain
[c] Instituto de Investigación en Ingeniería de Aragón (I3A), 50018 Zaragoza, Spain
[d] European Network of Excellence on High Performance and Embedded, Architecture and Compilation (HiPEAC)[1]

ARTICLE INFO

Article history:
Received 8 January 2010
Received in revised form 24 June 2010
Accepted 23 August 2010
Available online xxxx

Keywords:
WCET
Instruction cache-locking
Line-buffer

ABSTRACT

In multitasking real-time systems it is required to compute the WCET of each task and also the effects of interferences between tasks in the worst case. This is very complex with variable latency hardware, such as instruction cache memories, or, to a lesser extent, the line buffers usually found in the fetch path of commercial processors. Some methods disable cache replacement so that it is easier to model the cache behavior. The difficulty in these cache-locking methods lies in obtaining a good selection of the memory lines to be locked into cache. In this paper, we propose an ILP-based method to select the best lines to be loaded and locked into the instruction cache at each context switch (dynamic locking), taking into account both intra-task and inter-task interferences, and we compare it with static locking. Our results show that, without cache, the spatial locality captured by a line buffer doubles the performance of the processor. When adding a lockable instruction cache, dynamic locking systems are schedulable with a cache size between 12.5% and 50% of the cache size required by static locking. Additionally, the computation time of our analysis method is not dependent on the number of possible paths in the task. This allows us to analyze large codes in a relatively short time (100 KB with 10^{65} paths in less than 3 min).

1. Introduction

Real-time systems require that tasks complete their execution before specific deadlines. Given hardware components with a fixed latency, the worst case execution time (WCET) of a single task could be computed from the partial WCET of each basic block of the task. However, in order to improve performance, current processors perform many operations with a variable duration. This is mainly due to speculation (control or data) or to the use of hardware components with variable latency. Branch predictors fall in the first category, whereas memory hierarchy and datapath pipelining belong to the second one. A memory hierarchy made up of one or more cache levels takes advantage of program locality and saves execution time and energy consumption by delivering data and instructions with an average latency of a few processor cycles. Unfortunately, the cache behavior depends on past references and it is required to know the previous accesses sequence in order to compute the latency of a given access in advance. Resolving these *intra-task* interferences is a difficult problem its own. Anyway, real-time systems usually work with several tasks which may interrupt each other at any time. This makes the problem much more complex, since the cost of *inter-task* interferences must also be identified and bounded. Furthermore, both these problems cannot be accurately solved independently, since the path that leads to the WCET of an isolated task may change when considering interferences. Cache-locking tackles the whole problem by disabling the cache replacement, so the cache content does not vary. Specifically, for an instruction cache, the instruction fetch hits and misses depend on whether each instruction belongs to a cached and locked memory line and not on the previous accesses.

In this paper, we focus on the instruction fetch path and analyze several configurations of the memory architecture shown in Fig. 1. It consists of a line buffer (LB) and a lockable instruction cache (i-cache). The i-cache retains the fixed subset of instruction lines previously loaded by system software at task switches. It does not need fine-grained locking, but whole cache-locking. The LB has the size of a cache line and acts as a single-line cache memory regarding the tag, access latency and refill latency. The only difference with a conventional cache is that it prevents exploiting any

☆ This work was supported in part by grants TIN2007-66423 (Spanish Government and European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ICT 217068). This work is an extension of the technical report RR-09-01 with new contributions and experimental results [1].

* Corresponding author at: DIIS, Universidad de Zaragoza, 50018 Zaragoza, Spain.
E-mail addresses: luisapa@unizar.es (L.C. Aparicio), jsegarra@unizar.es (J. Segarra), clemente.rodriguezl@ehu.es (C. Rodríguez), victor@unizar.es (V. Viñals).

[1] http://www.hipeac.net/

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.08.008

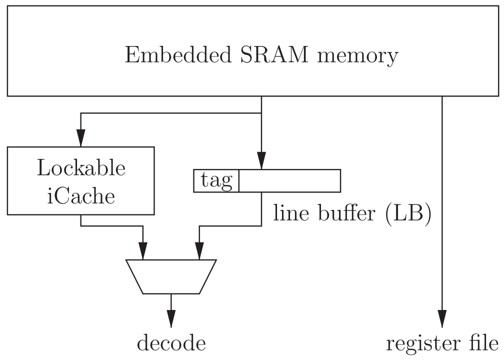


Fig. 1. Memory architecture considered.

temporal locality by invalidating its content on both i-cache hits and backward jumps to the currently buffered line. Instruction cache-locking can be implemented in several ways. One way is to design and synthesize a specific organization taking advantage of its control simplicity, since no replacement algorithm needs to be implemented and the refill happens always in bursts. Another way is making use of the locking capabilities present in many commercial processors devoted to the medium and high-end embedded market.[2]

We propose a new method intended to minimize the instruction cache contribution on the WCET. Previous works studying cache-locking behavior do not distinguish between spatial and temporal locality, so it is not clear how either of these affects the WCET. In order to evaluate the importance of spatial locality we first analyze an instruction line buffer (LB) working alone. Second, to analyze the impact on the WCET of exploiting temporal locality, we add an instruction cache, managed under static and dynamic locking, to the LB. In *static locking*, memory lines are preloaded at system start-up and remain unchanged during the whole system lifetime. We use the *Minimize Utilization* (Lock-MU) method for selecting the memory lines to be locked into the instruction cache [2]. In *dynamic locking*, the instruction cache is preloaded and locked in each context switch, so that there is one selection of memory lines per task. To obtain this selection of memory lines we propose *Maximize Schedulability* (Lock-MS), a new ILP-based method that considers both intra-task and inter-task interferences. That is, we get the selection of memory lines that provides the lowest overall execution cost (including preloading times) when used in a dynamic cache-locking multitasking system. We show how to model the system with easy to understand path-explicit constraints and then how to transform them into a compact model, which can be solved much faster.

This paper is organized as follows. In Section 2, we review the background and related work. Section 3 presents our path-explicit method for selecting the lines to be locked into the instruction cache. The model compaction is described in Section 4. Section 5 shows experiments comparing several selection procedures, memory architectures and analysis times. Finally, Section 6 presents our conclusions.

2. Related work

Multitask preemptive real-time systems must be schedulable to guarantee their expected operation. That is, all tasks must complete their execution before their deadline. Considering a *fixed priority* scheduler, feasibility of periodic tasks can be tested in a number of ways [3]. Response Time analysis is one of these mathematical approaches, and fits very well as a schedulability test. This approach is based on the following equation for independent tasks:

$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \qquad (1)$$

where R_i is the response time, C_i is the WCET and T_i is the period of each task i, respectively. It is assumed that tasks are ordered by priority (the lower the i the higher the priority). This equation provides the response time of each task after a few iterations and it has been used in previous studies [4–6]. A task t_i meets its real-time constraints if $R_i \leqslant D_i$, being D_i the deadline of the task.

The WCET of each task (C_i) is not easy to obtain in systems with cache memory. In the literature we find different methods to approach the WCET problem in the presence of caches [7]. These methods can be divided into those that analyze the normal behavior of the cache, and those which restrict its behavior to simplify the analysis.

The first kind of methods try to model each task and system as accurately as possible considering the dynamic behavior of the cache [8–15]. We compare our approach to one of these methods. A conventional cache analysis is very hard, since the worst path depends on the cache outcome, and the cache outcome affects the cost of each path. Due to this complexity, interferences among tasks are not usually considered and tasks are analyzed in isolation. This means that complementary methods are needed to adapt the WCET of each isolated task to multitasking systems. This may be done by further analysis to add the number of inter-task interferences and their cost to the cost of each task [5,6].

In turn, cache-locking methods restrict the cache behavior by using the ability to disable the cache replacement. Having specific contents fixed in cache, the timing analysis is easier, so these methods can afford a full system analysis, i.e., several tasks on a real-time scheduler. Cache-locking techniques can also be divided into *static* and *dynamic cache-locking*.

Static locking methods preload the cache content at system start-up and fix this content for the whole system lifetime so that it never gets replaced [2,16]. Martí Campoy et al. use a genetic algorithm to obtain the selection [16], whereas Puaut and Decotigny propose two low-complexity selection algorithms: one to minimize the utilization (Lock-MU) and another to minimize the interferences (Lock-MI) [2]. Lock-MI is no longer considered in similar studies, since Lock-MU always exhibits a better behavior. None of these three algorithms studies the possibility of worst path changes depending on the selected cache lines. Instead, the worst path is determined only once, assuming an empty cache. Afterward, these algorithms make a selection of lines to be locked, such that they optimize this path. These techniques are also called "single-path analyses" [17] and, as authors say, their approach is non-optimal. Studies comparing Lock-MU and the genetic algorithm approach conclude that their performance is very similar [18]. We use Lock-MU as a static locking reference, thus avoiding the possible dependencies on the initialization parameters that genetic algorithms may present. Also, we compare to a single-cycle fetch system (ideal performance bound).

Cache-locking approaches that disable cache replacements but allow the cache contents to be changed at run-time are known as dynamic locking methods [19–24]. Essentially, these methods differ in how they load the contents and lock the cache. The operating system may be used to change cache contents at context switches by means of a subroutine [19,20]. As an alternative, the content replacement can be launched by the operating system when a task reaches a certain program counter value [21]. Also,

[2] Mainstream Instruction Set Architectures supporting whole cache-locking include, for instance, ARM (ARM940), MIPS (Integrated Device Technology IDT79RC64xxx), Motorola 68K (Freescale Coldfire), Power (Freescale MPC74xx, MPC8540, PowerPC 440 core, e300 core). Other processors support partial cache-locking only (e.g. most ARM, IDT79RC46xx).

specific instructions may be used to control the cache replacement by tasks [22]. All these approaches call for a per-task selection of contents, with the drawback of repeatedly preloading the selection. Some of these methods have been improved by specific hardware designs [23]. Also, dynamic versions of the previous static locking methods have been proposed [24]. In this case, the idea is to divide a task into regions, each one with particular contents to be loaded and locked. So, automatically choosing the beginning and end of each region in an adequate way can be seen as a contribution which, in general, would benefit any content selection method. Besides, these versions merge region-locking with worst-path recomputing, reporting very high analysis times in such recomputation.

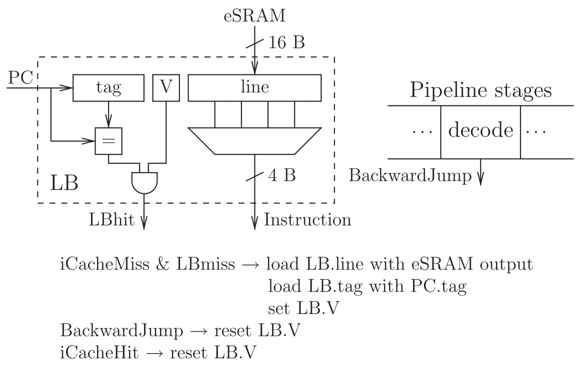
All previous static and dynamic cache-locking methods obtain the selection of lines to be locked by an heuristic search, either by greedy or genetic algorithms. So, their results may be dependent on the analyzed tasks and also on the search parameters. To prevent these factors, we compare our proposal to the ideal single-cycle fetch (no misses) plus the required preloading costs at context switches to always hit.

Our proposed ILP-based method, Lock-MS (Maximize Schedulability), is a dynamic cache-locking method that preloads the cache at context switches, and it is able to provide a solution that minimizes the worst overall cost in a feasible time.

Both locking and non-restrictive methods can be also improved by reducing or avoiding task interferences. Cache partitioning assigns a portion of the cache to each task, and restricts cache replacement to each individual partition [25–31]. Keeping the dynamic behavior within separate partitions, these techniques eliminate inter-task interferences and allow a separate analysis per partition. So, they are an interesting complement for other analysis methods. The drawback is that the reduction of the available cache size for each task may result in a worse performance than that of all tasks sharing the whole cache. The difficulty in cache partitioning is choosing the size of each partition. Once the partitions are set, the WCET analysis is equivalent to reducing the cache size to a partition size and also the number of tasks using each partition.

3. Selection of memory lines with Lock-MS (Max. Schedulability)

We consider a multitask system with a lockable set-associative instruction cache, so its behavior is completely predictable, avoiding both intra- and inter-task interferences. Having a dynamic locking approach, the cache content selected beforehand is preloaded every time a task starts/continues its execution in the CPU. In this way each task takes profit of the whole cache with the drawback of the preloading costs on context switches.



iCacheMiss & LBmiss → load LB.line with eSRAM output
load LB.tag with PC.tag
set LB.V
BackwardJump → reset LB.V
iCacheHit → reset LB.V

Fig. 2. LB organization and control.

Also, we consider that the memory system has a small instruction line buffer (LB) to capture spatial locality (see Fig. 1), i.e., a tagged buffer of the size of a cache line. The fetch lookup proceeds in parallel in both instruction cache and LB. A hit in either cache or LB delivers the requested instruction in a processor cycle. A miss in both structures leads to requesting the line to the next memory level (an embedded SRAM in our model), filling later the LB with the incoming line. The only specific behavior to consider is that the LB prevents any temporal locality exploitation (i.e., it self-invalidates its content on i-cache hits and in backwards jumps to the currently buffered line), which removes the potential dependencies on the previous path. Fig. 2 shows a way to organize the LB along with its control. Finally, we consider that our system has no additional sources of latency variability (data cache, branch predictor, out-of-order execution, etc.). Most embedded processors can operate under these considerations, which are also assumed in previous studies [2,16]. We also assume that all loop bounds are known.

The aim of our method is to provide, for each task, a selection of lines such that, when locked, the schedulability of the whole system is maximal. To obtain this selection of lines, Lock-MS considers the resulting WCET of each task, the effects of interferences between tasks and the cost of preloading the selected lines into cache. To get this selection we use *Integer Linear Programming* (ILP) [32,33]. Thus, our method is based on modelling all requirements as a set of linear constraints and solving the resulting system.

It is important to take into account that, using no component with a latency dependent on the previous path, there are no intra-task interferences in tasks, as observed in previous studies on cache-locking [34]. Adding the LB may seem to invalidate this statement but, with the previously defined behavior, the LB content depends exclusively on the location of the instruction executed before, so each possible path can be explicitly defined.

Problem formalization. Let us define a multitask system as a set of periodic tasks t_i, $1 \leqslant i \leqslant NTasks$. A task t_i can be modelled as a set of direct start-to-end paths $Path_{i,j}$, $1 \leqslant j \leqslant NPaths_i$. At the same time, each $Path_{i,j}$ contains a set of $NLines_{i,j}$ memory lines inside it.

Fig. 3(a) shows an example of a simple control flow. It shows several memory lines, L_1 to L_{12}, organized into basic blocks containing control structures and a function. Note that memory lines may be shared by different basic blocks (e.g. L_1, L_4) and basic blocks may contain several memory lines (e.g. leftmost path in the deepest loop: L_3L_4). The rightmost control flow corresponds to a function called from the two Ⓒ marks.

Fig. 3(b) shows an augmented representation of the control flow in Fig. 3(a), tailored to the subsequent WCET analysis. The new representation details three aspects: (i) Memory lines shared by different basic blocks have now been divided and assigned a unique identifier (e.g. $L_{4a}L_{4b}$). This allows us to account for a different cost and number of accesses on each part of the memory line. (ii) Function bodies must also be processed depending on the calling points. In Fig. 3(b) we have two analysis instances, called from memory lines $L_2(Ⓒ)^1$ and $L_{4b}(Ⓒ)^2$. (iii) To identify each start-to-end path ($Path_{i,j}$), path tags have been placed next to the basic blocks they traverse (e.g. path 2 goes across L_{1a}, L_{1b}, L_2, L_{9a}, L_{9b}, L_{11a}, $L_{11b} - L_{12a}$, L_{12b}, L_{8a}, L_{8b}). With the LB behavior defined above its content cannot be reused, i.e., if a memory line is needed after it has been consumed, it must be fetched again. This avoids the LB acting as a single-line cache. The consequence is that, on conditional branches, the worst-path will take any side of the branch but always the same one, since there is no dependence on the previous path. Also, the cost of a specific path traversing a specific memory line is constant, even if it is traversed multiple times (e.g. in path 2, L_{1b} is traversed $1 + bound_2^1$ times).

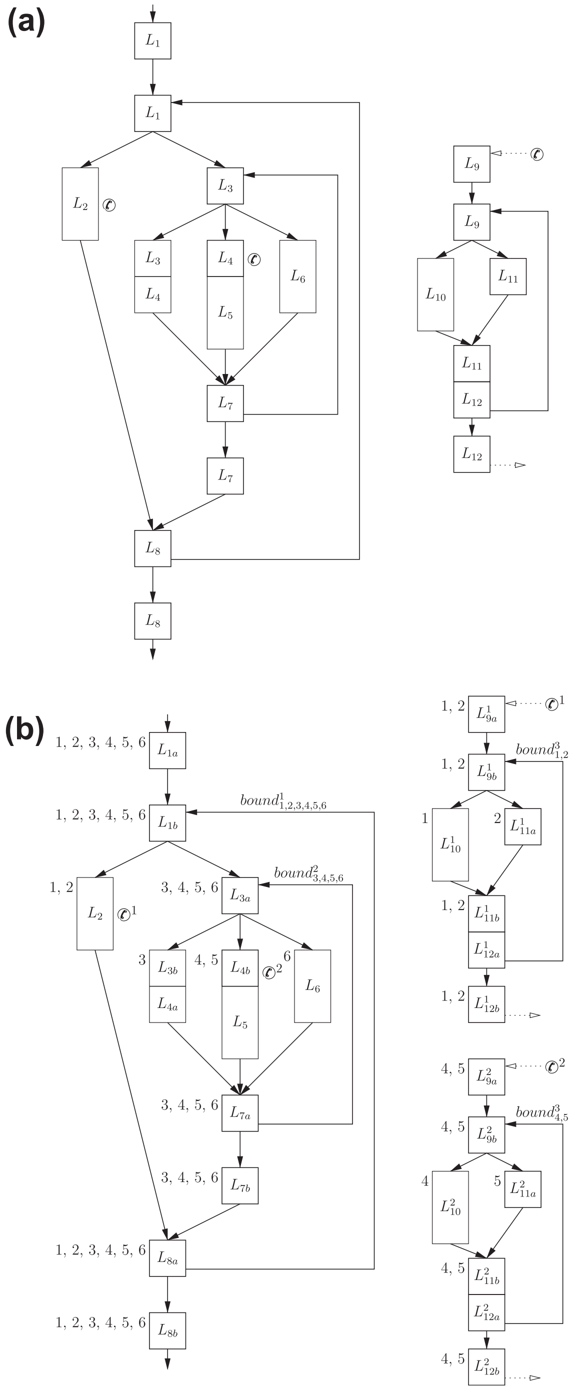


Fig. 3. Example of modelling a simple control flow. (a) Straightforward view: memory lines have been split to reflect basic block boundaries. (b) Augmented representation: (i) each (divided) memory line has a unique identifier, (ii) function instances are introduced, and (iii) basic blocks are annotated with the traversing path(s).

In this way, we can model each task t_i as a set of paths $Path_{i,j}$, each of them traversing its set of (parts of) memory lines (simply memory lines from now on). Thus, the cost of executing path j of task i ($PathC_{i,j}$) can be computed as the sum of the cost of traversing its instruction memory lines ($LC_{i,j,k}$):

$$PathC_{i,j} = \sum_{k=1}^{NLines_{i,j}} LC_{i,j,k}$$

For the path 2 of the task i in Fig. 3, it would be:

$$\begin{aligned} PathC_{i,2} &= \sum_{k=1}^{NLines_{i,2}} LC_{i,2,k} \\ &= LC_{i,2,1a} + LC_{i,2,1b} + LC_{i,2,2} + LC_{i,2,9a^1} + LC_{i,2,9b^1} + LC_{i,2,11a^1} \\ &\quad + LC_{i,2,11b^1} + LC_{i,2,12a^1} + LC_{i,2,12b^1} + LC_{i,2,8a} + LC_{i,2,8b} \end{aligned}$$

Cost of memory lines. Each memory line $L_{i,j,k} \in Path_{i,j}$ has an associated execution cost $LC_{i,j,k},\ 1 \leqslant k \leqslant NLines_{i,j}$ defined as follows:

$$LC_{i,j,k} = hitCost_{i,j,k} \cdot nhit_{i,j,k} + missCost_{i,j,k} \cdot nmiss_{i,j,k}$$

where *hitCost* and *missCost* are the execution costs on cache hit and miss, and *nhit* and *nmiss* are the number of cache hits and misses.

Let us now show how to compute each of these values. First, the number of cache hits and misses can be obtained by knowing the maximum number of accesses to each instruction memory line $nfetch_{i,j,k}$ through a given path j, and whether this line is cached or not:

$$nhit_{i,j,k} = nfetch_{i,j,k} \cdot cached_l$$
$$nmiss_{i,j,k} = nfetch_{i,j,k} - nhit_{i,j,k}$$

The binary variable $cached_l \in [0,1]$ indicates whether the memory line $L_{i,j,k} \in Path_{i,j}$ starting at the physical address $l \times memlineSize$ is cached and locked ($cached_l = 1$) or not ($cached_l = 0$). The constant $nfetch_{i,j,k}$ is the total number of times that this memory line is accessed within path j, including the loop iterations, e.g. $nfetch_{i,2,9b} = (1 + bound^1_2) \times (1 + bound^3_2)$. This constant is a loop bound, i.e., derived from source code annotations, abstract interpretation, or any equivalent methodology.

As an example of the constraints on the number of hits and misses, we show how to construct them on memory line L_{11} of path 2 in Fig. 3. This would be repeated for every memory line:

$$nhit_{i,2,11a^1} = nfetch_{i,2,11a^1} \cdot cached_{11}$$
$$nmiss_{i,2,11a^1} = nfetch_{i,2,11a^1} - nhit_{i,2,11a^1}$$
$$nhit_{i,2,11b^1} = nfetch_{i,2,11b^1} \cdot cached_{11}$$
$$nmiss_{i,2,11b^1} = nfetch_{i,2,11b^1} - nhit_{i,2,11b^1}$$

Cost of hits and misses. In order to compute the execution costs we assume a simple processor where instruction fetch and execution proceed sequentially. For a cached memory line containing one or more ($nIns_{i,j,k}$) instructions we can compute the total cost as the fetch cost plus the execution cost ($texec$). The fetch cost is the instruction cache memory hit time ($thit_{CM}$) times the number of fetched instructions ($nIns_{i,j,k}$) in $L_{i,j,k}$:

$$hitCost_{i,j,k} = texec_{i,j,k} + thit_{CM} \cdot nIns_{i,j,k}$$

Depending on the memory architecture, the penalties on cache misses may vary. Let us start assuming that we have a locked cache (replacements disabled) and no other component is present. In this case, the miss cost is the execution time of the instructions in the memory line, plus one cache miss time for each of them:

$$missCost_{i,j,k} = texec_{i,j,k} + tmiss_{CM} \cdot nIns_{i,j,k}$$

Adding a line buffer, the cache miss cost is the sum of the execution time *texec*, a single LB miss $tmiss_{LB}$ and one LB hit $thit_{LB}$ for each of the remaining instructions in the memory line ($nIns_{i,j,k} - 1$):

$$missCost_{i,j,k} = texec_{i,j,k} + tmiss_{LB} + thit_{LB} \cdot (nIns_{i,j,k} - 1)$$

Cache constraints. The number of memory lines that may be cached is limited by the cache configuration (number of sets S and ways W). This is modelled by the new constraints:

$$W \geqslant cached_s + cached_{s+S} + cached_{s+2S} + \cdots + cached_{s+nS}$$
$$\forall \quad 0 \leqslant s < S,\ s \in \text{Integer}$$

WCET. The WCET of a task i must be equal to the cost of its worst path, i.e., it must be equal or greater than any of its paths j:

$$wcet_i \geqslant Pathcost_{i,j} \quad \forall\ 1 \leqslant j \leqslant NPaths_i \tag{2}$$

Since we use a minimization model, this variable will be minimized and at the end we will obtain the minimum $wcet_i$ satisfying this constraint, which is the desired value.

Note that it is possible for the final selection to include memory lines of several alternative paths. This would be the case when having two very similar paths and thus, in order to reduce the WCET, the cost of both paths needs to be reduced.

Switch cost. When the goal is to obtain the plain WCET, the previous equations suffice. However, in order to take into account the context switches, their cost must be considered. Let us focus on the cost of a single context switch to task i. First, a constant cost *tswitch* for saving the state of the preempted task and restoring task i must be considered. The preloading time *tpreload* must also be added. We consider each preloading time equal to $tmiss_{CM} - thit_{CM}$, and it must be taken into account for each cached memory line ($cached_l = 1$). Finally, we must include one miss penalty to the LB for every context switch, since the worst case would be that the LB has been flushed during the context switch:

$$numcached_i = \sum_{l=0}^{Mlines-1} cached_l$$
$$tpreload_i = (tmiss_{CM} - thit_{CM}) \cdot numcached_i$$
$$switchCost_i = tswitch + tpreload_i + (tmiss_{LB} - thit_{LB})$$

where the constant *Mlines* is the number of physical memory lines in the system.

Minimization function: Worst overall cost. The worst overall cost of a task i is equal to the WCET plus the switch cost times the number of context switches $ncswitch_i$:

$$Wcost_i = wcet_i + ncswitch_i \cdot switchCost_i \tag{3}$$

Thus, minimizing $Wcost_i$ we obtain the selection of memory lines $cached_l$ (and the values of *wcet*, *switchCost*, etc.) such that when preloaded and locked into cache, the worst overall cost of task i is minimal.

Although the exact number of task switches $ncswitch_i$ (preemptions plus context switches on idle processor) is not known, it can be overestimated in many ways [5,6]. In our case, we can start with any upper bound. For instance, this number cannot be higher than the sum of the maximum number of invocations of any higher priority task during T_i:

$$ncswitch_i \leqslant \sum_{j=1}^{i-1} \left\lceil \frac{T_i}{T_j} \right\rceil \tag{4}$$

This is a pessimistic bound, but below we show how it can be improved, if needed. Note that, considering the cost of context switches, memory lines may be not cached even if they are used several times. For instance, think of a task using a memory line three times, so that at first glance it should be locked into cache. Now think of this task being preempted ten times. Clearly, the cost of preloading this line 10 times is higher than the cost of the three misses that we would get if this line was never locked into cache. With this added cost, the ILP model will consider the WCET of a task along with its context switch costs, so that it may choose not to cache and lock lines that result in a worse overall system execution time due to inter-task interferences.

Including high-level control information. Apart from structural control flow information, our program modelling can include additional constraints to deal with high level functional control flow information or execution-history dependent instruction timings. For instance, if it is known that a path inside a loop is taken once at most, or if taking a certain path implies a particular number of iterations, this could be added to the model by modifying the existing constraints on the definition of paths and number of line fetches, in a similar way to other ILP-based models [12].

For instance, in Fig. 4 (by Li et al. [12]) we can see a typical example with functional information. Let us assume that each box represents a different memory line. This example has two paths depending on the *if-then-else* construct, since clearly the loop must be assumed to take the maximum possible number of iterations (functionality constraint of iterations equal to 10). The interesting functional detail is that memory line B_5 can be only traversed once. Assuming that path $P1$ is always taking the *then* side and that path $P2$ is taking the *else* side once, the constraints for the maximum number of times that memory lines B_4 and B_5 are accessed could be expressed as:

$$nfetch_{i,P1,B_4} = 10$$
$$nfetch_{i,P1,B_5} = 0$$
$$nfetch_{i,P2,B_4} = 9$$
$$nfetch_{i,P2,B_5} = 1$$

Let us assume that, additionally, we know that, if B_5 is traversed, the number of iterations in the loop is exactly 5. In this case, the memory lines and values of *nfetch* (in brackets) for each path would be:

$$P1:\ B_1(1), B_2(11), B_3(10), B_4(10), B_5(0), B_6(10), B_7(1)$$
$$P2:\ B_1(1), B_2(6), B_3(5), B_4(4), B_5(1), B_6(5), B_7(1)$$

Response time. Once the ILP problem is solved, going back to Eq. (1) we have the WCET of each task, C_i, and the cost of preloading the cache with the selected lines. With these two values we can obtain the response time R_i to test the schedulability of the real-time system. A bound (tighter than that of Eq. (4)) of the maximum number of preemptions N_i can then be calculated as:

$$N_i = ncswitch_i \leqslant \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil$$

With the new N_i we can recalculate the ILP problem in an iterative way. We could also use more accurate and sophisticated bounds on

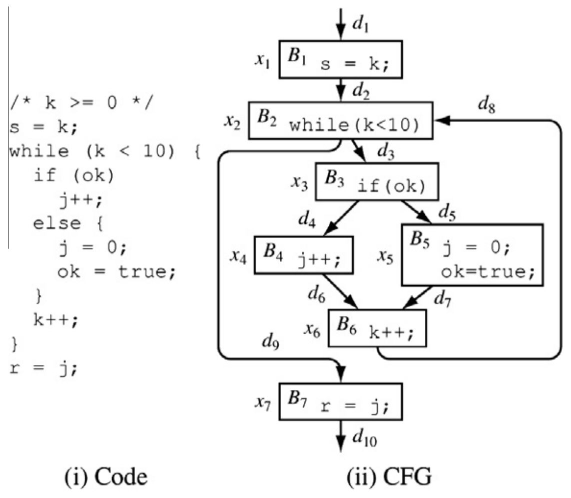

```
/* k >= 0 */
s = k;
while (k < 10) {
  if (ok)
    j++;
  else {
    j = 0;
    ok = true;
  }
  k++;
}
r = j;
```

Fig. 4. Example of a program with functional information on feasibility (see [12]).

the number of preemptions (e.g. [6]), but this is not the objective of the present paper.

In order to work with more accurate values, we simulate the multitasking system under the *rate monotonic* scheduler considering that all tasks are initially launched at the same time. Having the WCET and the number of cached lines of each task, the simulator schedules the tasks and provides the exact number of context switches N_i and response time R_i.

4. Avoiding the path-explicit constraints

Describing each path with a set of constraints implies a large ILP model when the number of possible paths grows, which in turn may decrease the solver speed. In this section we show that, by applying a transformation, our model is compacted and the description of the possible paths is no longer a limitation.

Let us take Fig. 5 as an example. It shows a simple control flow graph and its corresponding execution possibilities, depicting also the explosion of paths. In general, the WCET can be computed as the maximum of the four possible paths:

$$\begin{aligned}
P1 &= B0 + B1 + B3_1 + B4_1 + B6_{1,4}\\
P2 &= B0 + B1 + B3_1 + B5_1 + B6_{1,5}\\
P3 &= B0 + B2 + B3_2 + B4_2 + B6_{2,4}\\
P4 &= B0 + B2 + B3_2 + B5_2 + B6_{2,5}\\
WCET &= \max(P1, P2, P3, P4)
\end{aligned} \tag{5}$$

With an instruction locking cache whose content has been selected, several considerations must be done. First, Eq. (5) would compute the WCET considering all misses, e.g. selecting an empty cache. Our desired WCET is the minimum valid WCET considering any cache content k in the set of possible contents C:

$$WCET = \min_{k\in C}(\max(P1_k, P2_k, P3_k, P4_k)) \tag{6}$$

Our ILP-based method performs this computation. For a clearer notation, the k subindexes are not shown from now on:

$$WCET = \min(\max(P1, P2, P3, P4)) \tag{7}$$

Second, using a preloaded and fixed cache content, the cost of executing a block is independent of the previously executed blocks.[3] This idea can be easily extended to the LB, since its behavior depends on the previous executed instruction which is known for a given path. So, all Bi_{path} can be substituted by Bi and the common blocks can be computed in isolation. This allows us to rewrite Eq. (7) as:

$$\begin{aligned}
WCET &= \min(\max(B0 + B1 + B3 + B4 + B6,\\
&\quad B0 + B1 + B3 + B5 + B6, B0 + B2 + B3 + B4 + B6,\\
&\quad B0 + B2 + B3 + B5 + B6))\\
&= \min(B0 + B3 + B6 + \max(B1 + B4,\\
&\quad B1 + B5, B2 + B4, B2 + B5))
\end{aligned} \tag{8}$$

Fig. 6 shows an example of basic blocks and memory lines (L_x) to better see this simplification considering the LB. If a given memory line is selected to be cached and locked, its fetch cost will be always a cache hit. For non-cached memory lines, the fetch cost of their first reference must be the LB miss cost for every accessed line. The most interesting fetch costs appear when a memory line belongs to different basic blocks. This example includes all possible cases, namely, a memory line shared by a common basic block and then a conditional basic block (L_2), a memory line belonging to two alternative basic blocks (L_4) and a memory line shared by a conditional block and a common basic block (L_6). As explained above, these lines are divided and a particular cost is assigned to each part. Table 1 summarizes these LB costs for paths 1 and 2 of Fig. 6 assuming that none of these memory lines is cached. Using the path-explicit models for paths 1 and 2, there can be seen that divided memory lines consecutively accessed (e.g. L_{2a}, L_{2b}) have a single fetch cost. All this information can be obtained by statically analyzing the code.

In the column labeled as *Compact*, Table 1 shows the fetch cost of each memory line using the compact model. As it can be seen, when both Path 1 and Path 2 have the same fetch cost for a given line, it can be directly translated to the compact model (rule 1).

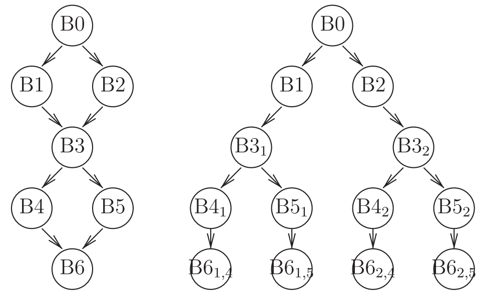


Fig. 5. Folded/unfolded control flow graph.

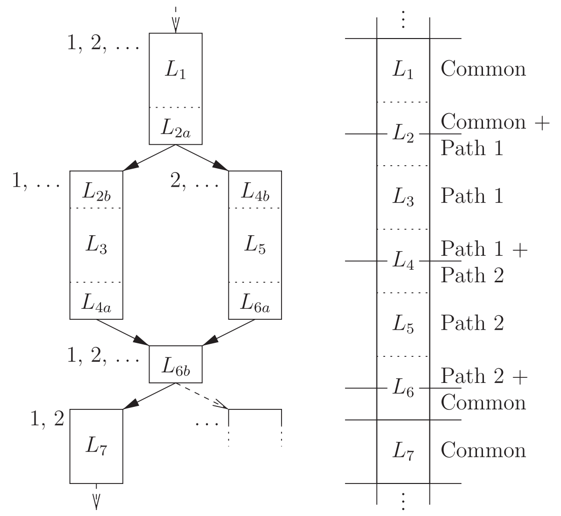


Fig. 6. Detailed example of path-explicit/compact fetch costs.

Table 1
Fetch cost (first reference) of memory lines in Fig. 6 assuming that they are not cached.

Line	Path-explicit		Compact
	Path 1	Path 2	
L_1	$tmiss_{LB}$	$tmiss_{LB}$	$tmiss_{LB}$
L_{2a}	$tmiss_{LB}$	$tmiss_{LB}$	$tmiss_{LB}$
L_{2b}	0	–	0
L_3	$tmiss_{LB}$	–	$tmiss_{LB}$
L_{4a}	$tmiss_{LB}$	–	$tmiss_{LB}$
L_{4b}	–	$tmiss_{LB}$	$tmiss_{LB}$
L_5	–	$tmiss_{LB}$	$tmiss_{LB}$
L_{6a}	–	$tmiss_{LB}$	0
L_{6b}	$tmiss_{LB}$	0	$tmiss_{LB}$
L_7	$tmiss_{LB}$	$tmiss_{LB}$	$tmiss_{LB}$

[3] This is true with the processor model assumed in this study. The application to processors including additional sources of time dependencies (such as branch predictor or out-of-order execution) may be addressed in future works.

Table 2
Classification of memory lines in start-to-end paths and common paths in Fig. 3.

Path 1	L_{1a}	L_{1b}	L_2									L_{8a}	L_{8b}	L^1_{9a}	L^1_{9b}	L^1_{10}		L^1_{11b}	L^1_{12a}	L^1_{12b}
Path 2	L_{1a}	L_{1b}	L_2									L_{8a}	L_{8b}	L^1_{9a}	L^1_{9b}		L^1_{11a}	L^1_{11b}	L^1_{12a}	L^1_{12b}
Path 3	L_{1a}	L_{1b}		L_{3a}	L_{3b}	L_{4a}				L_{7a}	L_{7b}	L_{8a}	L_{8b}							
Path 4	L_{1a}	L_{1b}		L_{3a}			L_{4b}	L_5		L_{7a}	L_{7b}	L_{8a}	L_{8b}	L^2_{9a}	L^2_{9b}	L^2_{10}		L^2_{11b}	L^2_{12a}	L^2_{12b}
Path 5	L_{1a}	L_{1b}		L_{3a}			L_{4b}	L_5		L_{7a}	L_{7b}	L_{8a}	L_{8b}	L^2_{9a}	L^2_{9b}		L^2_{11a}	L^2_{11b}	L^2_{12a}	L^2_{12b}
Path 6	L_{1a}	L_{1b}		L_{3a}					L_6	L_{7a}	L_{7b}	L_{8a}	L_{8b}							
CmnAll	L_{1a}	L_{1b}										L_{8a}	L_{8b}							
*Cmn*1, 2			L_2											L^1_{9a}	L^1_{9b}			L^1_{11b}	L^1_{12a}	L^1_{12b}
*Cmn*3, 4, 5, 6				L_{3a}						L_{7a}	L_{7b}									
*Cmn*4, 5							L_{4b}	L_5						L^2_{9a}	L^2_{9b}			L^2_{11b}	L^2_{12a}	L^2_{12b}

When only one of the paths traverses a memory line, the fetch cost can also be safely translated to the compact model (rule 2). These two rules can be applied to the non-divided memory lines L_1, L_7 and L_3, L_5, respectively. The cost translation of the divided memory lines (L_2, L_4, L_6) is not so straightforward. For lines whose parts belong to alternative paths (L_{4a} and L_{4b}), the fetch costs can also be directly translated (rule 2) since, being alternative, they are never used together. For the lines belonging to a common path and also a non-common path (L_2, L_6), the full fetch cost must be associated to the part in the common path (L_{2a} and L_{6b}) and 0 to the particular path (L_{2b} and L_{6a}) (rule 3). With these translation rules, the cost of any path (i.e., the sum of the cost of its lines) is the same as the sum of the same lines in the compact model. That is, the sum of column Path 1 (or Path 2) in Table 1 is the same as the sum of the same lines in column Compact. Thus, for any memory line, we have an equivalent representation of fetch costs which is independent of the number of paths and does not add any overestimation.

Going back to Eq. (8), we can obtain equivalent expressions of the max function as follows. If $B1 > B2$, then $B1 + B4 > B2 + B4$, so clearly $B2 + B4$ could be dismissed. Otherwise, $B1 + B4 \leqslant B2 + B4$ and $B2 + B4$ must remain in the expression. So, using the max operator first on the $B1$, $B2$ pair and then on the $B4$, $B5$ pair we have:

$$\begin{aligned}\max(B1 + B4, B1 + B5, B2 + B4, B2 + B5) &= \max(\max(B1, B2) + B4, \max(B1, B2) + B5)\\ &= \max(\max(B1, B2) + \max(B4, B5))\\ &= \max(B1, B2) + \max(B4, B5) \qquad (9)\end{aligned}$$

Thus, we can rewrite Eq. (8) as:

$$WCET = \min(commonCost + \max(B1, B2) + \max(B4, B5)) \qquad (10)$$

As a more complex example consider the control flow in Fig. 3. Instead of having explicit constraints for the paths, the WCET is constructed as the sum of common costs plus the maximum of any alternative path. Table 2 shows the lines traversed by each path in Fig. 3, and how these lines are grouped into common paths. This table can be translated into a tree (Fig. 7). This tree can be seen as a CFG where memory lines appear only once, i.e., each node is a set of common lines and each branch is an alternative path. Following Fig. 7, the constraints are:

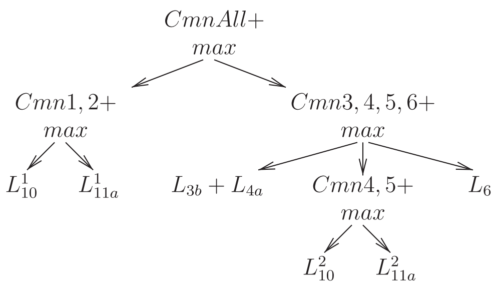

$$\begin{tikzcd}[column sep=small] & & & CmnAll+\\max \arrow[dll] \arrow[drr] & & \\ & Cmn1,2+\\max \arrow[dl] \arrow[dr] & & & & Cmn3,4,5,6+\\max \arrow[dl] \arrow[d] \arrow[dr] & \\ L^1_{10} & & L^1_{11a} & & L_{3b}+L_{4a} & Cmn4,5+\\max \arrow[dl] \arrow[dr] & L_6 \\ & & & & L^2_{10} & & L^2_{11a}\end{tikzcd}$$

Fig. 7. Compact constraint graph of Fig. 3.

$$\begin{aligned}
&wcet_i = CmnAll + ForkAll\\
&CmnAll = LC_{1a} + LC_{1b} + LC_{8a} + LC_{8b}\\
&ForkAll \geqslant Cmn1,2 + Fork1,2\\
&ForkAll \geqslant Cmn3,4,5,6 + Fork3,4,5,6\\
&Cmn1,2 = LC_2 + LC^1_{9a} + LC^1_{9b} + LC^1_{11b} + LC^1_{12a} + LC^1_{12b}\\
&Fork1,2 \geqslant LC^1_{10}\\
&Fork1,2 \geqslant LC^1_{11a}\\
&Cmn3,4,5,6 = LC_{3a} + LC_{7a} + LC_{7b}\\
&Fork3,4,5,6 \geqslant LC_{3b} + LC_{4a}\\
&Fork3,4,5,6 \geqslant Cmn4,5 + Fork4,5\\
&Fork3,4,5,6 \geqslant LC_6\\
&Cmn4,5 = LC_{4b} + LC_5 + LC^2_{9a} + LC^2_{9b} + LC^2_{11b} + LC^2_{12a} + LC^2_{12b}\\
&Fork4,5 \geqslant LC^2_{10}\\
&Fork4,5 \geqslant LC^2_{11a}
\end{aligned}$$

These constraints compute the same $wcet_i$ as Eq. (2). However, they use directly the line cost variables $(LC_{i,j,k})$ without needing the explicit path cost constraints *Pathcost*. Thus, these new constraints can be substituted on the previous path-explicit model to avoid the explicit definition of paths.

Furthermore, an hybrid path-explicit/compact model is also possible. It could provide an accurate WCET analysis on specific code parts (e.g. unfeasible paths or execution-history dependent instruction timings) and a fast analysis on compactable parts. This would allow to tune the WCET analysis so that the accuracy is adapted to different code parts.

5. Performance evaluation

We assume periodic tasks with fixed priorities managed by a *Rate Monotonic* scheduler. Table 3 shows the two sets of tasks used in our experiments. Benchmarks include JPEG integer implementation of the forward DCT, CRC, matrix multiplication, integral computation by intervals, matrix inversion, computation of roots of quadratic equations and FFT. Sources have been compiled with GCC 2.95.2-O2. The WCET in this table refers to the LB-only system and it has been computed without context switches. Periods have been set so that the CPU utilization of each task in the LB-only system is 1.2. The "small" and "medium" task sets and the relation between the periods for each task have already been used in previous studies [2].

Note that the WCETs and periods of each task set follow different patterns. In the small task set, WCETs and periods grow as the

Table 3
Task sets “small” and “medium”.

Set	Task	LB-only WCET	Period	Size
Small	jfdctint	10,108	23,248	1072 B
	crc	109,696	329,088	536 B
	matmul	542,229	2,440,031	208 B
	integral	716,633	3,583,165	400 B
Medium	minver	8522	19,601	1360 B
	qurt	10,117	30,351	752 B
	jfdctint	10,108	44,475	1072 B
	fft	2,886,680	15,010,736	1016 B

priority in tasks decreases. It can be seen that, in this task set, periods grow by an order of magnitude approximately in most cases. However, the medium task set has relatively small WCETs and periods for all tasks but the one with the lowest priority, which is around three orders of magnitude larger. This means that the lowest priority task in the medium task set will be interrupted many times. So, in general, the medium task set will have more context switches than the small task set for a given time period.

The target architecture considered in our experiments is an ARM7 processor with instructions of 4 bytes. The LB size (and memory/cache line size) is 16 bytes, or 4 instructions. The instruction caches are varied in size from 128 bytes to 4 KB and the eSRAM is kept fixed at 256 KB. In order to compute memory delays we have used Cacti v5.3, a cache circuit modelling tool [35], assuming a future high-performance embedded processor built in 32 nm technology and running at processor cycle equivalent to 36 FO4.[4] We have verified that all the tested caches, excluding the fully associative ones, meet the cycle time constraint. Besides, the access time of the 256 KB eSRAM is 7 cycles if we choose to implement it with low standby power transistors. Therefore, instruction fetch costs are 1 cycle on cache or LB instruction hits and 8 cycles on LB misses. All data accesses are delivered directly from the eSRAM. Thus, the modelled execution costs are 1 cycle for non-executed predicated instructions,[5] 2 cycles for non-memory instructions, and 1 + 7 cycles for loads and stores.

Next we characterize separately each single task by computing its WCET in three fetch systems, namely, Direct-eSRAM fetch (upper bound), LB-only fetch, and single-cycle fetch (lower bound). Then, we study the whole multitasking system acting on the memory system of Fig. 1 (LB + iCache system) and compare Lock-MS (multiple per-task selections) to Lock-MU (a single selection for all the tasks) [2]. Next, we discuss the analysis cost of the compact Lock-MS model. Finally, we compare to a worst-case conventional cache analysis.

5.1. Spatial locality

Fig. 8 shows, for each benchmark, the WCET of the LB-only and single-cycle systems relative to a Direct-eSRAM system. A single-cycle fetch system would correspond to a system with an ideal instruction cache, whereas a Direct-eSRAM fetch system would correspond to an easily predictable system with the instruction cache disabled. As it can be seen, WCETs are significantly lowered just by exploiting the spatial locality. On average, an LB-only system reduces the Direct-eSRAM WCET by a 0.53× factor. In turn, a single-cycle system reduces the Direct-eSRAM WCET by a 0.33× factor on average. Obviously this single-cycle system would need the whole program preloaded and fitting into the cache, but it gives

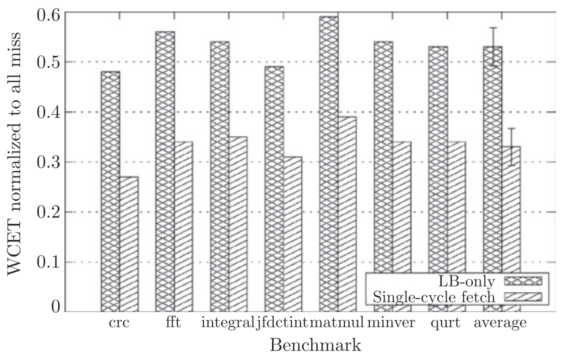


Fig. 8. WCET of LB-only and single-cycle fetch systems relative to fetching directly from the eSRAM.

an idea of how far the ideal case is. So, the WCET reduction achieved by enhancing an LB-only system with an instruction cache (temporal locality) could be up to an additional speed-up of 1.61 at most (from 0.53 to 0.33, assuming that the ideal 0.33 is reachable). Results for individual tasks depart somewhat from the average case, but the trends are quite similar.

Utilization can be used to test the spatial locality effects on multitasking systems. It is computed as the fraction of time the CPU is busy executing the task set in the worst case:

$$U = \sum_{i=1}^{NTasks} \frac{Wcost_i}{T_i}$$

Utilization values above 1 indicate the system is not executable (and not schedulable). On values below 1, the system may (or may not) be schedulable, depending on the response times and deadlines ($R_i \leqslant T_i,\ \forall i$). As indicated above, the periods have been set for both task sets in order to get a utilization value of 1.2 in the LB-only system. In contrast, if all fetches were directly delivered from the eSRAM the utilization would raise up to 2.33 and 2.24 for the small and medium task sets, respectively, whereas with ideal single-cycle fetch the utilization decreases up to 0.75 and 0.72, respectively.

In conclusion, the instruction LB is beneficial for hard real-time systems because it improves both WCET and processor utilization at a very low cost, but significant room for improvement still remains, which can be partly capitalized by a carefully managed instruction cache.

5.2. Dynamic vs. static locking

In this section, we compare Lock-MS against Lock-MU in the LB plus iCache system. Lock-MS selects for each task separately the lines to be dynamically loaded at each context switch, whereas Lock-MU computes a single, static selection for the whole task set [2]. There has been no address tuning of the tasks, i.e., the code of each task begins at an address mapped to cache set 0.

This comparison cannot be made with WCETs only, since the periods and priorities of tasks lead to a different number of context switches when combined with the WCETs. In general, shorter WCETs cause fewer preemptions, which in turn decrease the system response time. Thus, since the cost associated to context switches depends on the locking method, it is necessary to test how all these parameters affect the final system. First, we test whether the system is schedulable. If so, we obtain the response time of the task with the lowest priority (t_4). As a simple speed-up metric, we divide the period of this task by its response time: T_4/R_4. This metric gives an idea of the looseness of the schedule. Fig. 9 shows results with different cache configurations: from

[4] A fan-out-of-4 (FO4) represents the delay of an inverter driving four copies of itself. A processor cycle of 36 FO4 at 32 nm technology would result in a clock frequency around 2.4 GHz, which is in line with the market trends [36].

[5] Predicated instructions are those general (not jump) instructions that include several test bits, so that the instruction is executed as long as the test is true.

128 bytes to 2 KB and from 256 bytes to 4 KB for the small and medium task sets, respectively. The percentage of the instruction cache size with respect to the task set size is shown on top of each cache size group. As it can be seen, Lock-MS gets a schedulable system with a cache of just 5% of the code size, whereas Lock-MU is not schedulable until the cache size reaches significantly higher percentages (40% and 10% for the small and medium task sets, respectively). Additionally, for all cache sizes but the largest ones (80%) the speed-up obtained with Lock-MS is always above that of Lock-MU in both task sets. This is an indicator that Lock-MS may achieve a pretty good performance with simple hardware.

The response time speed-ups of an ideal instruction cache (single-cycle fetch) are 2.28 and 3.14 for the small and medium task sets, respectively. Notice that this ideal system does not incur in context switch penalties. In order to compare with an easily reproducible and hardware independent dynamic locking policy, we consider the ideal single-cycle fetch (no misses) plus the required preloading costs at context switches to always hit. The response time speed-ups of single-cycle plus context switch penalties are 2.19 and 2.55 for the small and medium task sets, respectively. So, context switch penalties are responsible for 4% (small task set) and 19% (medium task set) of the speed-up differences between these two systems. To give insight into such differences, we have computed the number of context switches per system response time (i.e., launching all tasks at the same time, the number of context switches until the CPU becomes idle, divided by the time required for this to happen). On average, the number of context switches per system response time for the medium task set is 2.27 times higher than for the small task set. Obviously, the number of context switches affects Lock-MS and single-cycle with penalties, but it does not affect Lock-MU (static-locking) nor single-cycle without penalties.

Another interesting detail is that Lock-MS is not sensitive to associativity in most cases. Among all experiments, only two intermediate cache sizes in the medium task set (512 bytes and 1 KB) take profit from a large associativity. This means that Lock-MS can use fast and simple direct mapped caches with no performance loss. On the contrary, depending on the code placement, Lock-MU may be specially sensitive to the number of ways (e.g. see the steep steps leading from direct mapping to 2-way and 4-way in the 80% groups in both task sets). This happens when several tasks have a high usage of memory lines mapped to the same cache line, but not all of them can be locked at the same time. Increasing associativity may reduce or solve the problem but at the cost of increasing hit time,[6] area and energy consumption. Performing a previous optimization of code layout should also reduce this problem for Lock-MU, but it is not needed for Lock-MS.

For large cache sizes (80%), it can be seen that static approaches may offer better results than a dynamic approach since the cache preload penalty grows, as can be seen in Fig. 9. That is, when all significant memory lines (or the whole tasks) almost or completely fit in cache, using a unique selection for the whole task set should be better than using a selection per task. This has been already observed in other studies [37].

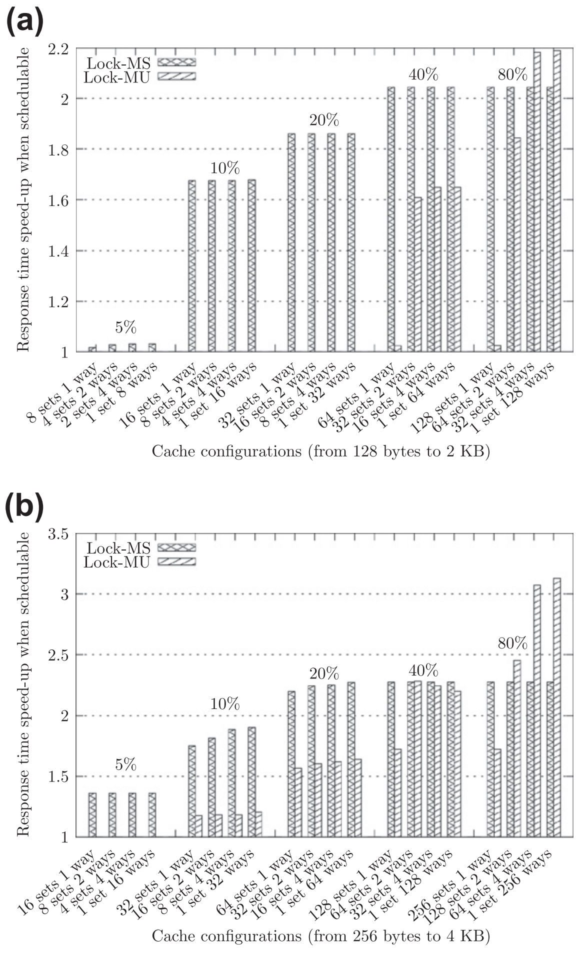


Fig. 9. Response time speed-ups. (a) Small task set. (b) Medium task set.

5.3. Analysis cost

The computational cost of Lock-MS depends on the analyzed tasks and the employed ILP solver. Its application to the tasks in Table 3 takes a few milliseconds, so its discussion is not relevant. To get a better insight into the analysis cost of Lock-MS we have designed a collection of synthetic tasks. These tasks have between 16 KB and 96 KB of code size, with up to 2^{216} possible paths through consecutive if-then-else constructs. So, we can test Lock-MS on very large tasks designed specifically to be difficult to analyze. Every task is analyzed in the LB plus iCache system for three caches of growing sizes, totalizing 33 experiments. Table 4 summarizes the experimentation space. For these experiments we have used *lp_solve* version 5.5.0.14 with the default options on a 2 GHz, 64-bit Intel Xeon.

First of all, we have observed that the solver gets the optimal (or a very close) solution of the minimization function *Wcost* (Eq. (3)) in a very short time, and then spends the rest of the time testing similar solutions. Thus, using the first integer solution can save us much time and its accuracy can be easily tested by the real (not integer) solution. Although the real solution is not necessarily a valid ILP solution, it is faster (the solution space is continuous) and there cannot exist any better solution to the problem. Thus, the difference between any valid integer solution and the real one gives an idea of the goodness of the integer solution. Fig. 10 shows the cumulative distribution of these differences in our experiments. The *x*-axis shows the differences and the *y*-axis shows the number of occurrences. It can be seen that in 28% of cases the difference is 0, i.e., the real solution is also integer, and both solutions coincide. Also, there are no differences above 0.45%. This means that the obtained result is exact in 28% of cases, and in the other 72% the possible overestimation when using the first integer solution differs at most five CPU cycles per thousand. However, since the real solutions may not be valid, lower differ-

[6] In our experiments we have not modelled the hit time increase of fully associative caches to better show the trends. However, if we were designing a real system we could use Cacti in order to compute the multicycle hit time, run again Lock-MS and get a more realistic speed-up.

Table 4
Experimentation space.

Task		Instruction cache	
Size (KB)	N. Paths	Sizes (KB)	Sets × Ways
16	2^{36}	4, 8, 12	64 × 4, 8, 12
32	2^{72}	8, 16, 24	64 × 8, 16, 24
48	2^{108}	12, 24, 36	64 × 12, 24, 36
64	2^{144}	16, 32, 48	64 × 16, 32, 48
96	2^{9}, 2^{18}, 2^{30}, 2^{54}, 2^{108}, 2^{162}, 2^{216}	24, 48, 72	64 × 24, 48, 72

ences may not be reachable, so it is perfectly possible that, even with a 0.45% difference, a first integer solution is the best one.

Fig. 11 depicts the analysis time (both for the real and first integer solutions) as the complexity of the analysis grows. The *x*-axis shows the task size. Each task has been analyzed for three cache sizes, which can be seen in the plot as three marks for the same *x* value, both for the real and integer solutions. The resulting values have been fitted to a potential function, also shown in the figure. It can be seen that the analysis time grows approximately in a quadratic way as the complexity (code size, cache size and number of paths) grows simultaneously.

Fig. 12 shows a similar study, but varying just the number of paths from 2^9 to 2^{216}. In this case we focus on the 96 KB task with cache sizes of 24, 48 and 72 KB (64 sets of 24, 48 and 72 ways, respectively). It can be seen that there is no ascending trend across the broad range of tested number of paths. This means that the number of paths does not limit the analysis time when using the compact model of Lock-MS. Thus, the results are affected by the particular relation between the memory lines and cache lines of each problem. Anyway, the analysis of such large benchmarks

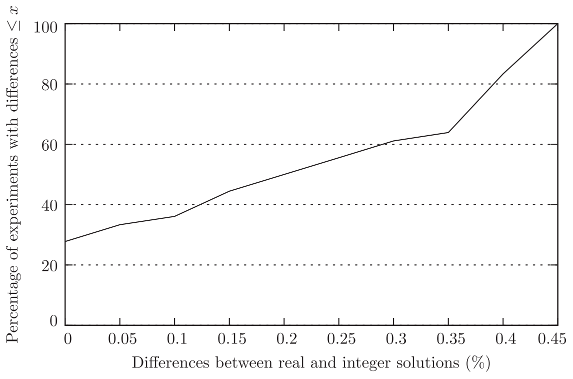


Fig. 10. Cumulative distribution of differences between the real and integer solutions.

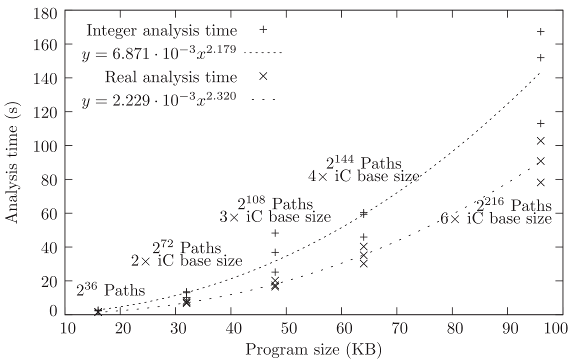


Fig. 11. Analysis time as the complexity grows (iCache base sizes of 4, 8 and 12 KB).

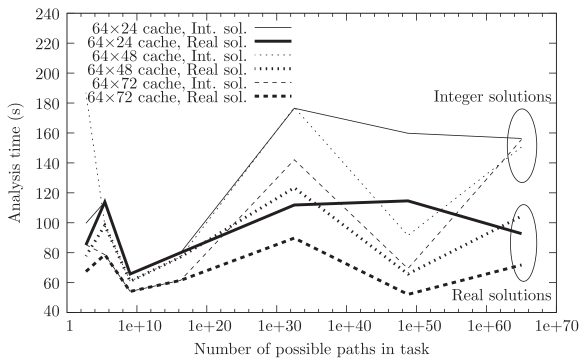


Fig. 12. Analysis time varying the number of conditional statements.

(96 KB, 10^{65} paths, 72 way set-associative cache) takes no longer than 3 min. This analysis time is relatively short compared to other ILP-based WCET analysis models, which have been criticised for being too large and hard to analyze [12,38].

5.4. Lock-MS vs. worst-case conventional cache

As pointed out in the previous section, Lock-MS has a relatively low computation cost compared to methods that analyze the WCET assuming a conventional cache behavior [8,38]. Additionally, such methods analyze the WCET of single tasks, so further analysis/overestimation on inter-task interferences is required in order to provide safe system bounds.

Given that the tasks in Table 3 are not specially hard to analyze, in order to provide additional comparisons we have obtained the response time speed-ups using a conventional cache by a worst-case simulation method [8]. This technique has a high computational cost because it explores all required paths by a branch and bound algorithm, but it provides a very accurate WCET bound for each task. The inter-task interferences have been accounted as an additional cost for worst case evictions, i.e. the minimum of task lines and cache lines, times the miss penalty for every context switch. Our results show that WCETs using conventional instruction caches are very similar to those with Lock-MS (differences between −3.8% and 7.4%). When considering inter-task interferences in the system, both approaches present similar results. Fig. 13 shows the response time speed-ups for the small task set using direct-mapped caches having the same sizes than previous experiments. Both Lock-MS and the worst-case conventional cache behavior have low sensitivity to associativity, so these values are not presented. Also, both behaviors reach a similar saturation point, where additional cache size does not entail higher speed-ups. Lock-MS performs better than the worst-case conventional cache bound on very small caches (5–10% of code size). This is due to the high intra-task interferences in small conventional caches, whereas Lock-MS avoids them by locking. The rest of experiments (20–80% of code size) show similar speed-ups (differences lower than 5%). These trends using a conventional cache appear also for the medium task sets. However, in this case the system is schedulable with large caches only (40–80%) and with speed-ups lower than 1.1 due to the number of context switches being much higher.

In general, methods using locking caches seem more adequate for complex systems. When the analysis is affordable both with locking and non-locking methods, several points must be considered. First, conventional caches do not resolve intra-task interferences, which may be an important drawback on small caches. For larger caches, the bound on the number of preemptions may become a key factor. In Lock-MS the cost of a context switch is determined by the selected contents, and only those contents that

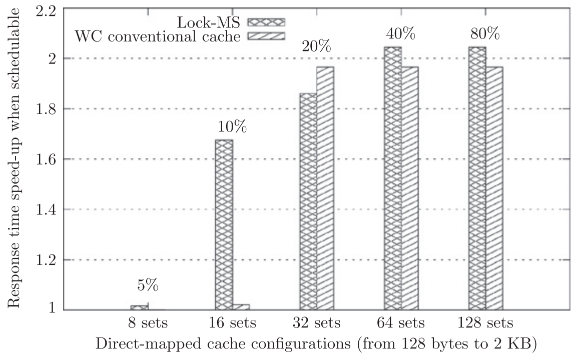


Fig. 13. Lock-MS vs. worst-case conventional cache behavior for the small task set.

are worth to be cached are selected. On the contrary, using a conventional cache, context switches do not imply specific costs but overestimations, which may be higher and difficult to bound accurately. These overestimations may be addressed by further inter-task interference analysis, although it would increase the analysis time.

6. Conclusions

In this paper, we have proposed an ILP-based method (Lock-MS) to obtain the selection of memory lines to be loaded and locked into an instruction cache at each context switch in multitasking real-time systems. This selection takes into account the requirements of each single task and also the effects of interferences between tasks. Additionally, we show that the description of the model can be compacted when the number of possible paths grows. The combination of both the path-explicit and compact model is also possible, which is interesting for including high-level control flow information.

To isolate the benefits of the instruction cache we analyze first three cacheless options: either fetching directly from the eSRAM, from a line buffer or from an ideal system providing single-cycle fetch. Compared to the direct eSRAM fetch, the line buffer reduces processor utilization by almost 50% at a very low cost, and we take this option as a suitable baseline for the following experiments.

We also compare Lock-MS, which provides a per-task memory line selection, with an existing static locking approach (Lock-MU) for an instruction fetch architecture having a line buffer and a cache. Lock-MS performs better than Lock-MU for cache sizes under 40% of the code size, improving schedulability with small caches. With larger caches, most or all the highly accessed code can be cached and static-locking performs better, since it does not suffer from cache reloading penalties. Our results also show that our approach is not sensitive to the cache configuration (sets vs. ways) but to the total cache size, being able to successfully exploit direct mapped caches.

In order to show that Lock-MS is computationally feasible we have designed a collection of synthetic tasks designed specifically to be large and difficult to analyze, using them in a set of large caches. We find that the solver obtains a very good solution in a very short time, and the remaining time is spent discarding very similar solutions. To assess the quality of the first integer solutions we use the real solution, showing that it is optimal in around 25% of cases and, in the rest of cases, no more than 0.45% larger than the real solution, which can be taken as an overestimation bound. Using the compact Lock-MS representation we show that the analysis time grows approximately in a quadratic way with respect to the problem complexity (essentially the task size), with the number of paths in the task having very little impact. This allows us to analyze large codes in a relatively short time (100 KB with 10^{65} paths in less than 3 min).

Finally, comparing Lock-MS with a worst-case conventional cache analysis, both options perform in a similar way in WCET. Regarding the response time in multitask, Lock-MS performs better and has higher predictability.

References

[1] L.C. Aparicio, J. Segarra, V. Viñals, C. Rodríguez, J.L. Villarroel, Improving the use of instruction cache-locking methods in multitasking real-time systems, Tech. Rep. RR-09-01, Universidad de Zaragoza, March 2009.
[2] I. Puaut, D. Decotigny, Low-complexity algorithms for static cache locking in multitasking hard real-time systems, in: IEEE Real-Time Systems Symposium, 2002.
[3] L. Sha, T. Abdelzaher, K.-E. Arzétn, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, A.K. Mok, Real time scheduling theory: a historical perspective, Real-Time Systems 28 (2004) 101–155.
[4] J.V. Busquets, A. Wellings, Adding instruction cache effect to schedulability analysis of preemptive real-time systems, in: Proceedings of RTAS96, 1996.
[5] C. Lee, K. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, C. Kim, Bounding cache-related preemption delay for real-time systems, IEEE Transactions on Software Engineering 27 (9) (2001).
[6] J. Staschulat, S. Schliecker, R. Ernst, Scheduling analysis of real-time systems with precise modelling of cache related preemption delay, in: Proceedings of the Euromicro Conference on Real-Time Systems, 2005, pp. 41–48.
[7] R. Wilhelm et al., The determination of worst-case execution times – overview of the methods and survey of tools, ACM Transactions on Embedded Computing Systems (TECS).
[8] L.C. Aparicio, J. Segarra, C. Rodríguez, J.L. Villarroel, V. Viñals, Avoiding the WCET overestimation on LRU instruction cache, in: IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008.
[9] R. Arnold, F. Mueller, D. Whalley, M. Harmon, Bounding worst-case instruction cache performance, in: IEEE Real-Time Systems Symposium, 1994, pp. 172–181.
[10] C. Ferdinand, R. Wilhelm, Efficient and precise cache behavior prediction for real-time systems, Real-Time Systems 17 (2–3) (1999) 131–181.
[11] C. Healy, D. Whalley, M. Harmon, Integrating the timing analysis of pipelining and instruction caching, in: IEEE Real-Time Systems Symposium, 1995, pp. 288–297.
[12] Y.T.S. Li, S. Malik, A. Wolfe, Cache modelling for real-time software: beyond direct mapped instruction caches, in: IEEE Real-Time Systems Symposium, 1996, pp. 254–264.
[13] T. Lundqvist, P. Stenström, An integrated path and timing analysis method based on cycle-level symbolic execution, Real-Time Systems 17 (2–3) (1999) 183–207.
[14] F. Mueller, Timing analysis for instruction caches, Real-Time Systems 18 (2–3) (2000) 217–247.
[15] H. Theiling, C. Ferdinand, R. Wilhelm, Fast and precise WCET prediction by separated cache and path analyses, Real-Time Systems 18 (2–3) (2000) 157–179.
[16] A. Martí Campoy, N. Perles Ivars, J.V. Busquets Mataix, Static use of locking caches in multitask preemptive real-time systems, in: IEEE Real-Time Embedded System Workshop, 2001.
[17] H. Falk, S. Plazar, H. Theiling, Compile-time decided instruction cache locking using worst-case execution paths, in: CODES + ISSS '07: Proceedings of the Fifth IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, ACM, New York, NY, USA, 2007, pp. 143–148. <http://www.doi.acm.org/10.1145/1289816.1289853>.
[18] A. Martí Campoy, I. Puaut, N. Perles Ivars, J.V. Busquets Mataix, Cache contents selection for statically-locked instruction caches: an algorithm comparison, in: Euromicro Conference on Real Time Systems, IEEE Computer Society, Los Alamitos, CA, USA. <http://www.doi.ieeecomputersociety.org/10.1109/ECRTS.2005.34>.
[19] A. Martí Campoy, E. Tamura, S. Sáez, F. Rodrfguez, J.V. Busquets Mataix, On using locking caches in embedded real-time systems, in: ICESS, 2005, pp. 150–159.
[20] A. Martí Campoy, N. Perles Ivars, V.J. Busquets Mataix, Static use of locking caches in multitask preemptive real-time systems, in: Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium), 2001.
[21] A. Arnaud, I. Puaut, Dynamic instruction cache locking in hard real-time systems, in: Proceedings of the 14th International Conference on Real-Time and Network Systems (RNTS), Poitiers, France, 2006.
[22] P. Jain, S. Devadas, D. Engels, L. Rudolph, Software-assisted cache replacement mechanisms for embedded systems, in: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, IEEE Press, San Jose, CA, USA, 2001, pp. 119–126.
[23] E. Tamura, J.V. Busquets Mataix, A. Martí Campoy, Towards predictable, high-performance memory hierarchies in fixed-priority preemptive multitasking real-time systems, in: International Conference on Real-Time and Network Systems.

[24] I. Puaut, WCET-centric software-controlled instruction caches for hard real-time systems, Real-Time System Euromicro Conference (2006) 217–226.

[25] R. Reddy, P. Petrov, Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems, in: CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, ACM, New York, NY, USA, 2007, pp. 198–207. <http://www.doi.acm.org/10.1145/1289881.1289917>.

[26] V. Suhendra, T. Mitra, Exploring locking and partitioning for predictable shared caches on multi-cores, in: DAC '08: Proceedings of the 45th Annual Conference on Design Automation, ACM, New York, NY, USA, 2008, pp. 300–303. <http://www.doi.acm.org/10.1145/1391469.1391545>.

[27] D. Chiou, P. Jain, S. Devadas, L.Rudolph, Application-specific memory management of embedded systems using software-controlled caches, Tech. Rep., MIT, November 1999.

[28] P. Ranganatham, S. Adve, N. Jouppi, Reconfigurable caches and their application to media processing, in: Proceedings of the Annual International Symposium on Computer Architecture, 2000, pp. 214–224.

[29] J.E. Sasinowski, J.K. Strosnider, A dynamic programming algorithm for cache memory partitioning for real-time systems, IEEE Transactions on Computer 42 (8) (1993) 997–1001. <http://www.dx.doi.org/10.1109/12.238493>.

[30] G.E. Suh, L. Rudolph, S. Devadas, Dynamic cache partitioning for simultaneous multithreading systems, in: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, 2001, pp. 116–127.

[31] C. Zhang, F. Vahid, W. Najjar, A highly configurable cache architecture for embedded systems, in: Proceedings of the Annual International Symposium on Computer Architecture, 2003, pp. 136–146.

[32] V. Chvátal, Linear Programming, W.H. Freeman & Company, 1983.

[33] F. Rossi, P.V. Beek, T. Walsh, Handbook of Constraint Programming, Elsevier, 2006.

[34] A. Martí Campoy, S. Sáez, N. Perles Ivars, J.V. Busquets Mataix, Performance comparison of locking caches under static and dynamic schedulers, in: Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, IFAC/IFIP/IEEE, Lagow, 2003.

[35] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, N.P. Jouppi, Cacti 5.3. <http://www.hpl.hp.com/research/cacti/>.

[36] Chart watch: high-performance embedded processor cores, Microprocessor Report 22 (2008) 26–27.

[37] A. Martí Campoy, N. Perles Ivars, F. Rodrfguez, J.V. Busquets Mataix, Static use of locking caches vs. dynamic use of locking caches for real-time systems, in: Canadian Conference on Electrical and Computer Engineering, 2003.

[38] R. Wilhelm, Why AI + ILP is good for WCET, but MC is not, nor ILP alone, in: VMCAI, 2004, pp. 309–322.

Luis C. Aparicio received the degree in Mathematics from University of Zaragoza (Spain) in 1997. In 2000, he joined this University and currently he is a Ph.D. student. His current research interests are cache memories in real-time systems, timing analysis and worst-case execution time.

Juan Segarra graduated in Computer Science from Universitat Jaume I (Spain) and hold his Ph.D. in 2003 from the same university. Also in this year, he joined the University of Zaragoza, where he is currently working as a lecturer in the Informática e Ingeniería de Sistemas Department. Also, he is member of the Computer Architecture group (gaZ) of the University of Zaragoza. His research interests include QoS, media distribution and also worst-case execution time and worst-case memory performance in hard real-time systems.

Clemente Rodríguez received the Bachelor's degree in Computer Science from the Autonomous University of Barcelona, Spain in 1981 and the degree in Computer Science in 1986 from the Polytechnic University of Catalonia, Spain. He is a professor at the Computer Architecture and Technology Department of the Basque Country University where he has been working since 1988. Since 1990 he leads a group in parallel architectures and computer architectures which has been participating in several European and local research projects. Since 2006 he is a member of the Computer Architecture Group of the University of Zaragoza. His interests include timing analysis and memory hierarchy.

Víctor Viñals received the M.S. degree in Telecommunication, and the Ph.D. degree in Computer Science from the Universitat Politècnica de Catalunya (UPC) in 1982 and 1987, respectively. He was associate professor in the Facultat d'Informática de Barcelona (UPC) in the 1983–1988 period. Currently, he is full professor in the Informática e Ingeniería de Sistemas Department at the University of Zaragoza, in Zaragoza (Spain). His research interests include processor microarchitecture, memory hierarchy and parallel computer architecture. He is member of the ACM and the IEEE Computer Society. He also belongs to the Juslibol Midday Runners Team and to the Computer Architecture Group of the University of Zaragoza.