Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems*

Isabelle Puaut David Decotigny
IRISA, Campus universitaire de Beaulieu,
35042 Rennes Cédex, France
Email: {puaut/ddecotig}@irisa.fr

Abstract

Cache memories have been extensively used to bridge the gap between high speed processors and relatively slow main memories. However, they are a source of predictability problems because of their dynamic and adaptive behavior, and thus need special attention to be used in hard-real time systems. A lot of progress has been achieved in the last ten years to statically predict the worst-case behavior of applications with respect to caches in order to determine safe and precise bounds on tasks worst-case execution times (WCETs) and cache-related preemption delays. An alternative approach to cope with caches in real-time systems is to statically lock their contents such that memory access times and cache-related preemption times are predictable. In this paper, we propose two low-complexity algorithms for selecting the contents of statically-locked caches. We evaluate their performances and compare them with those of a state of the art static cache analysis method.

1 Introduction

Extensive studies have been performed on schedulability analysis to guarantee timing constraints in hard real-time systems. Many of these schedulability analysis methods assume that the worst-case execution times (WCETs) of the tasks are known. While most of them consider that the cost of task preemption is zero to simplify the analysis, some other methods account for task preemption costs (context switches [KAS93], cache-related preemption delays [BN94, BMSO+96]).

Many methods have been proposed in the last ten years in order to determine the WCETs of tasks in a safe but not overly pessimistic manner, based on a static analysis of the tasks source and/or object code [PB00]. Such *static WCET analysis methods* achieve a high-level analysis of the tasks, considered in isolation, to identify the worst execution path inside a task, and a low-level analysis to compute safely and precisely the worst-case execution times of program fragments on a given processor. A large number of researches have been undertaken on low-level analysis to deal with the increasingly complex processor micro-architectures (presence of instruction and data caches [Mue00, LS99], pipelines [HAM+99], branch predictors [CP00]).

1.1 Caches in real-time systems

Caches are small and fast buffer memories used to speed up the memory accesses. They contain the memory blocks that are likely to be accessed by the CPU in the near future. They are a very effective means of speeding up the memory accesses in the average case. However, the worst-case behavior of applications, which is of prime importance in hard real-time systems, is hard to predict in a safe and precise way in the presence of caches, due to the presence of intra-task and inter-task interferences. *Intra-task* interferences occur when a task overrides its own blocks in the cache due to conflicts for cache blocks. *Inter-task* interferences arise in multitasking systems because of the preemptions and imply a so-called *cache-related preemption delay* to reload the cache after a task is preempted. This predictability issue may lead the designers of hard real-time systems to choose not to use caches, or to use on-chip static RAM (scratchpad memories) [ABS01].

However, two classes of approaches, described hereafter, can be used to deal with caches in hard real-time systems.

Cache analysis methods. A first class of approaches to deal with caches in hard real-time systems is to use them without any restriction, and resort to *static analysis* techniques to predict their worst-case impact on the system schedulability.

*23rd IEEE International Real-Time Systems Symposium, Austin, TX, USA, December 2002.

At the intra-task level, static WCET analysis techniques achieve a classification of the memory accesses regarding the instruction or data caches (e.g. *hit* when it can be proved that a memory access always results in a cache hit, *miss* otherwise). Establishing such a classification requires special attention for the schedulability analysis not to be overly pessimistic. Indeed, the simple approach that considers that every access to memory results in a cache miss causes the tasks WCETs to be largely overestimated, which may cause the schedulability analysis to fail while the system may actually be feasible. The classification methods can be based on data-flow analysis [Mue00], abstract interpretation [AFMW96], integer linear programming techniques [LMW96], or symbolic execution [LS99].

At the inter-task level, work has been undertaken to obtain safe and precise estimates of the cache-related preemption delays [LHS+98]. In this work, at every possible preemption point, the blocks that will be used by each task after that point are determined by static analysis, thus avoiding to consider that the whole memory accessed by the task has to be reloaded in the cache after a preemption.

Task partitioning and cache locking. A second class of approaches to deal with caches in real-time systems is to use them in a restricted or customized manner, so as to adapt them to the needs of hard real-time systems and schedulability analysis.

Cache partitioning techniques (e.g. [Kir89, SS93]) assign reserved portions of the cache (partitions) to certain tasks in order to guarantee that their most recently used code or data will remain in the cache despite preemptions. The dynamic behavior of the cache is kept within partitions. These techniques eliminate the inter-task interferences, but a way to tackle intra-task interferences (cache-aware WCET analysis) is still required.

Another way to deal with caches in real-time systems is to use *cache locking techniques*, which load the cache contents with some values and lock it in order to ensure that the contents will remain unchanged [CIBM01]. This ability to lock cache contents is available in several commercial processors (Motorola ColdFire MCF5249, Motorola PowerPC 440, Motorola MPC7451, MIPS32, ARM 940 and ARM 946E-S for example). The cache contents can be loaded and locked at system start for the whole system lifetime (*static cache locking*), or changed during the system execution, like for instance when a task is preempted by another one (*dynamic cache locking*). The key property of cache locking techniques is that the time required to access the memory is predictable.

1.2 Paper contents and contributions

This paper explores the use of *static cache locking* of *instruction caches* in multitasking real-time systems. Static cache locking is attractive from several points of views. First of all, it improves the system performance compared to a system that does not use caches, with respect to both average and worst-case system performance.

In addition, with static cache locking, the time required to perform a memory access is *predictable* (it is either a *hit* or a *miss* depending on whether the value is locked in the cache or not). While WCET analysis is still required, using static cache locking alleviates the need for using complex static analysis techniques for computing WCETs and cache-related preemption delays, and results in more simple WCET analysis tools. In particular, it eliminates the issue of integrating cache analysis techniques with those for other architectural features (pipelines, branch predictors, etc). Static cache locking can also be used when no cache analysis method can apply, due for instance to non-deterministic or poorly documented cache replacement strategies (e.g. pseudo-random replacement policies).

Another important benefit of static cache locking is that the technique addresses both intra-task and inter-task interferences, which is unique among the cache management techniques presented above.

Furthermore, since static cache locking schemes statically partition the cache blocks among tasks, the cache-related preemption delay is null (or is constant and equal to the time required to reload the processor prefetch buffer if the processor is equipped with such an architectural feature). This characteristic is particularly important for large caches (see Section 4).

Finally, implementing cache locking turns out to be a light task once the contents of the locked cache are selected, provided that appropriate hardware support is available. No modification of the compilation tool chain is required to implement static cache locking. In particular, the addresses of values (instructions and data structures) need not be modified. To be implemented, static cache locking only requires to execute a small routine at the system start-up to load the contents of the cache with the selected values, and lock the cache.

However, statically locking the contents of instruction caches raises the issue of selecting the cache contents. In the framework of hard real-time systems, the main objective of the algorithms for cache contents selection is to improve the worst-case system behavior according to some of the metrics used by schedulability analysis methods, such as CPU utilization or interferences between tasks. The main issue is then to avoid performing an exhaustive search of all the possible cache contents, which would require an intractable computation cost. For instance, for a direct-mapped cache,

if up to 4 blocks of code can be mapped into every cache block, checking the feasibility of the system with all possible cache contents would require 4^B feasibility tests, with B the number of cache blocks. This complexity led Campoy *et al.* in [CIBM01] to use a genetic algorithm for the selection of the cache contents.

In this paper, we propose two algorithms to select the contents of the instruction cache for periodic task sets. In contrast to [CIBM01], the proposed algorithms select the contents of the locked cache in a *non blind manner*, by using the tasks memory access patterns of the instruction flow. An important property of both algorithms is their *low complexity* (pseudo-polynomial). Both algorithms optimize the worst-case behavior of the task set: the first one aims at minimizing the CPU utilization of the task set, and does not prescribe any particular schedulability analysis method, whereas the second one, designed for fixed-priority schedulers, aims at reducing the interferences between tasks due to preemptions. The worst-case performances of benchmark systems using the proposed cache contents selection algorithms are analyzed. Another contribution of this work is a comparison of the performance of task sets using static cache locking with a state of the art static cache analysis technique, so as to experimentally assess the domains of applications of both classes of approaches.

1.3 Paper organization

The remainder of the paper is organized as follows. Section 2 introduces the assumptions and notations used throughout the paper. Section 3 then presents the two algorithms to select the contents of the statically locked cache so as to optimize the system schedulability. The performance of these algorithms is analyzed in Section 4, in which we compare the performance of task sets using the proposed static cache locking algorithms with their performance using a state of the art static cache analysis technique. We conclude in Section 5 with a summary of the paper contributions and directions for future work.

2 Assumptions and notations

This section introduces the assumptions and notations used in the paper concerning the target architecture (§ 2.1) and the task model (§ 2.2). Background on cache-aware schedulability analysis is also presented (§ 2.3).

2.1 Architecture model

We consider a CPU equipped with a W-ways set-associative instruction cache with a LRU (Least Recently Used) replacement policy. The cache is of size S_c and comprises a total of B fixed-size blocks of size S_b ($S_c = B * S_b$). Blocks are grouped into S *sets* of W cache blocks; an instruction at address *ad* is loaded into one of the W blocks of set $\lfloor \frac{ad}{S_b} \rfloor mod\ S$ (see figure 1).

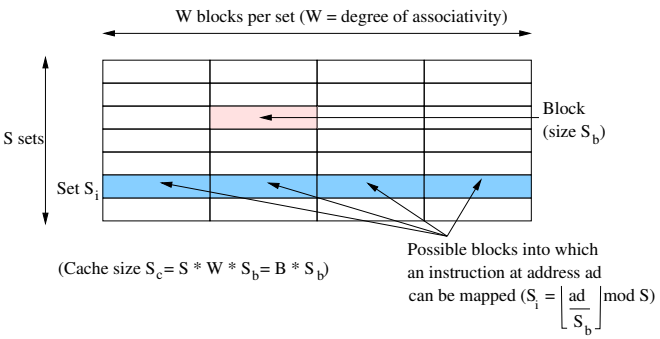

Figure 1. Cache model

Note that our model encompasses *direct-mapped* caches ($W = 1, S = B$), *set-associative* caches ($1 < W < B$) and *fully-associative caches* ($W = B, S = 1$).

In order to allow static cache locking to be implemented, we assume that there exists a mechanism to *lock* a block into the instruction cache, meaning that the cache replacement algorithm will never operate on that block. Such a mechanism is available in several commercial processors (e.g. Motorola PowerPC 440, Motorola MPC7451, ARM 940T and ARM 946E-S).

In addition to the instruction cache, the CPU may include a prefetch buffer of size S_b to improve sequential accesses to instructions. When an instruction is to be executed, it is first searched for in the prefetch buffer before searching for it in the memory hierarchy: in the instruction cache first, and then, in case of a cache miss, in the main memory.

We assume that an instruction takes t_{hit} clock cycles to execute if it is in the instruction cache or in the prefetch buffer, and $t_{miss} > t_{hit}$ otherwise.

2.2 Task model

We consider a set of N periodic tasks $T_{i,\,1 \leq i \leq N}$, with periods P_i and deadlines equal to periods. The worst-case execution time of task T_i is noted C_i.

The code of a task T_i is split into *program lines* $L_{i,j}$ of size S_b; a *program line* is a sequence of instructions mapped into a cache block.

We further assume in a first approximation that the sequence of instructions accessed by a task T_i on its worst-case execution path is known[1]. Let $WSeq_i$ be the sequence of program lines $L_{i,j}$ along the worst-case execution path, an occurrence of $L_{i,j}$ in the sequence meaning that an instruction of program line $L_{i,j}$ is executed. For example,

[1] We discuss the validity of this assumption in paragraph 3.3.

$WSeq_i = L_{i,1}L_{i,1}L_{i,1}L_{i,3}L_{i,2}L_{i,1}$ means that three instructions of $L_{i,1}$ are executed, followed by an instruction of $L_{i,3}$, one instruction of $L_{i,2}$ and an instruction of $L_{i,1}$.

Let σ_i be $WSeq_i$ in which any consecutively repeated element in the sequence is discarded (e.g. if we consider the previous example, $\sigma_i = L_{i,1}L_{i,3}L_{i,2}L_{i,1}$). This transformation of $WSeq_i$ into σ_i accounts for the presence of the prefetch buffer; if no prefetch buffer is present, then $\sigma_i = WSeq_i$. Finally, concerning a task T_i, let $n_{load_{i,j}}$ be the number of accesses to the memory hierarchy (instruction cache or main memory) concerning program line $L_{i,j}$ on the worst-case execution path. For instance, if $WSeq_i = L_{i,1}L_{i,1}L_{i,1}L_{i,3}L_{i,2}L_{i,1}$, then $\sigma_i = L_{i,1}L_{i,3}L_{i,2}L_{i,1}$ and $n_{load_{i,1}} = 2$ if there is a prefetch buffer, while $\sigma_i = WSeq_i$ and $n_{load_{i,1}} = 4$ otherwise.

2.3 Schedulability analysis

The algorithms introduced in this paper aim at minimizing metrics used by two cache-sensitive schedulability analysis methods for periodic task sets, named respectively CUA (Cache-aware Utilization-based Analysis) for static and dynamic priority scheduled systems, and CRTA (Cache-aware Response Time Analysis) for static priority scheduled systems. These two methods take into consideration the cache-related preemption delay. In the sequel, this delay is named γ_i and refers to an upper bound on the delay that a task T_i imposes on the tasks preempted by T_i.

CUA is an extension [BN94] of the well known Liu and Layland's [LL73] utilization-based schedulability conditions, that takes the cache-related preemption delay into account. The utilization ratio given in equation 1 below represents the proportion of the processor used in task computations.

$$U = \sum_{i=1}^{N} \frac{C_i + \gamma_i}{P_i} \quad (1)$$

For dynamic priority preemptively scheduled systems (*e.g.* Earliest Deadline First), CUA corresponds to the necessary and sufficient feasibility condition $U \leq 1$. For static priority systems with priorities assigned according to the rate monotonic policy, CUA corresponds to condition 2 below.

$$U \leq N * (2^{\frac{1}{N}} - 1) \quad (2)$$

Condition 2 is a sufficient condition only, and hence can be too pessimistic. To overcome this problem, response time analysis (RTA) methods [JP86, TBW94] give a necessary and sufficient condition for static priority scheduled systems without caches, and have been extended in [BMSO+96] to take the cache-related preemption delay into account. The extended schedulability condition is named CRTA. For a task T_i, the principle of CRTA is to consider the interferences produced by the execution of the higher priority tasks on T_i on an increasing time window w_i^n. The response time R_i of task T_i is the fixed point of the sequence given in equation 3 below, where $hp(T_i)$ denotes the set of tasks that have a higher priority than T_i.

$$\begin{aligned} w_i^0 &= C_i \\ w_i^{n+1} &= C_i + \sum_{T_j \in hp(T_i)} \left\lceil \frac{w_i^n}{P_j} \right\rceil * (C_j + \gamma_j) \rightarrow R_i \end{aligned} \quad (3)$$

For a task T_i, this series converges (to T_i's response time, R_i) when $\sum_{T_j \in hp(T_i) \cup \{T_i\}} \frac{C_j + \gamma_j}{P_j} \leq 1$. R_i can then be compared against T_i's deadline P_i to determine T_i's schedulability.

3 The algorithms

This section introduces two algorithms to select the contents of the locked instruction cache. Both algorithms are greedy (they do not reconsider the assignment of a cache block once it has been decided) and have a pseudo-polynomial complexity. They use the knowledge of the memory accesses made by the tasks along their worst-case execution paths (sequences σ_i). The two algorithms are based on different metrics to select the cache contents in order to optimize the task set schedulability. The first one aims at minimizing the worst-case CPU utilization, as defined in equation 1, while the second one aims at minimizing the interferences between tasks, as defined in equation 3. It is important to note that the metrics used for the selection of the locked cache do not prescribe the use of any particular schedulability analysis method: although CUA and CRTA are used to evaluate the performance of the algorithms in section 4, any scheduling policy taking into account the cache-related preemption delay can be used for schedulability analysis.

In the following, we term *statically locked cache* a cache for which every block is locked with zero or one program line: $Cache = array[0..S-1][0..W-1]$ of program lines.

3.1 Algorithm Minimize Utilization (Lock-MU)

This first algorithm, named Lock-MU (for *Minimize Utilization*) selects the contents of the locked cache so as to minimize the CPU utilization of the task set as defined in equation 1. The only pieces of information required to apply Lock-MU are the number of accesses to the memory hierarchy concerning every program line along the worst-case execution path ($n_{load_{i,j}}$) and the task periods (P_i). Since Lock-MU minimizes the CPU utilization of the task set, it is particularly suited to scheduling policies based on the CPU utilization.

Algorithm description. Let L_s be the set of program lines (for all possible tasks) that can be mapped into the W blocks of set s. The algorithm locks into the cache the W program lines $L_{i,j}$ of L_s having the highest $\frac{n_{load_{i,j}}}{P_i}$ ratio.

Complexity. Let η_s be the number of program lines (for all possible tasks) that can be mapped into the W blocks of set s ($\eta_s = \mid L_s \mid$). The algorithm requires that the program lines of every L_s be sorted by decreasing $\frac{n_{load_{i,j}}}{P_i}$ ratio, which is achieved in $O(S * \eta_s * log(\eta_s))$ with S the number of sets. For every set s, $O(W)$ operations are then required to select the highest element of this set, with W the degree of associativity.

Optimality. It can be shown that the algorithm is optimal with respect to the minimization of the CPU consumption when the worst-case execution sequences σ_i are known.

Sketch of proof. With the notations introduced in section 2, we can express C_i as follows:

$$C_i = \Big(\sum_{L_{i,j} \in \sigma_i} n_{load_{i,j}} * t_{i,j} \Big) + \Big(\mid WSeq_i \mid - \mid \sigma_i \mid \Big) * t_{hit}$$
$$= \Big(\sum_{L_{i,j} \in \sigma_i} n_{load_{i,j}} * t_{i,j} \Big) + constant$$

with $t_{i,j}$ the time required to execute an instruction $L_{i,j}$ depending on the contents of the locked cache (t_{hit} if $L_{i,j} \in Cache$, t_{miss} otherwise).

Since for a task T_i, γ_i and P_i are independent of the cache contents, minimizing $\sum_{i=1}^{N} \frac{C_i + \gamma_i}{P_i}$ is equivalent to minimizing $\sum_{i=1}^{N} \frac{C_i}{P_i}$. With the expression of C_i given above, this comes down to minimizing $\sum_{i=1}^{N} \sum_{L_{i,j} \in \sigma_i} \frac{n_{load_{i,j}} * t_{i,j}}{P_i}$.

If program lines are separated according to the sets into which they are mapped, minimizing the utilization is equivalent to minimizing:

$$\sum_{i=1}^{N} \sum_{s=0}^{S-1} \sum_{\substack{L_{i,j} \in \sigma_i, \\ \lfloor \frac{ad(L_{i,j})}{S_b} \rfloor mod S = s}} \frac{n_{load_{i,j}} * t_{i,j}}{P_i} =$$
$$\sum_{s=0}^{S-1} \sum_{i=1}^{N} \sum_{\substack{L_{i,j} \in \sigma_i, \\ \lfloor \frac{ad(L_{i,j})}{S_b} \rfloor mod S = s}} \frac{n_{load_{i,j}} * t_{i,j}}{P_i}$$

since the first two sums are independent (in the above formulas, $ad(L_{i,j})$ denotes the address of $L_{i,j}$).

Minimizing the utilization is then equivalent to minimizing, for every set s:

$$\sum_{i=1}^{N} \sum_{\substack{L_{i,j} \in \sigma_i, \\ \lfloor \frac{ad(L_{i,j})}{S_b} \rfloor mod S = s}} \frac{n_{load_{i,j}} * t_{i,j}}{P_i}$$

W blocks have to be assigned in set s. It comes from last formula that minimizing the utilization is equivalent to locking in the cache (i.e. setting $t_{i,j} = t_{hit}$) the W program lines with the highest ratio $\frac{n_{load_{i,j}}}{P_i}$, thus showing the optimality of the algorithm with respect to the minimization of the CPU utilization.

3.2 Algorithm Minimize Interferences (Lock-MI)

Our second algorithm for cache contents selection, named Lock-MI, for *Minimize Interferences*, selects the cache contents so as to minimize, for every task T_i, the interference (as defined in equation 3) that T_i has on the lower priority tasks. Like algorithm Lock-MU, Lock-MI selects the cache contents based on the knowledge of the worst-case sequence of instructions σ_i. The pseudo-code of algorithm Lock-MI is given in figure 2.

```
1  for s:=0 to S - 1 do
2     for w:=0 to W - 1 do
3        Cache[s][w] := ∅;
4     end
5  end
6  for i:=1 to N do
7     Ci := Estimate_WCET(Cache);
8  end
9  for s:=0 to S - 1 do
10    for w:=0 to W - 1 do
11       for each (i,j) such that Li,j can be mapped into Cache[s][w] do
12          gain_Ci := n_load_i,j * (t_miss − t_hit);
13          decrease_i,j := 0;
14          for all Tt ∈ lpe(Ti) do
15             decrease_i,j := decrease_i,j +
                  (Rt{C1,...,Ci,...,CN} − Rt{C1,...,Ci − gain_Ci,...,CN}) / Pt;
16          end
17       end
18       select Li,j such that decrease_i,j is maximized
19       Cache[s][w] := Li,j;
20       Ci := Ci − gain_Ci;
21    end
22 end
```

Figure 2. Source code of algorithm Lock-MI

Algorithm description. First, lines 1 to 5 initialize the cache with nothing locked in. Then, lines 6 to 8 compute an estimation of the WCET of every task T_i with this initial

cache contents. Every C_i is obtained by static WCET analysis (in the experiments described in section 4, we use the Heptane static WCET analysis tool).

Lines 9 to 22 compute for every cache block $Cache[s][w]$ the program line to be locked. This is done by computing for every candidate program line $L_{i,j}$ the interference decrease regarding the other lower priority tasks T_t incurred when locking $L_{i,j}$ in the cache. According to the number of times $L_{i,j}$ is loaded from the memory hierarchy ($n_{load_{i,j}}$), the decrease of C_i induced by locking $L_{i,j}$ in the cache is computed (variable gain_C_i, line 12). Then, for every task T_t with a priority lower or equal than the one of T_i (noted $T_t \in lpe(T_i)$), the response time R_t of task T_t is evaluated using CRTA (equation 3). This evaluation is achieved with and without locking $L_{i,j}$ in the cache. The decreases of response times for all T_t induced by the locking of $L_{i,j}$ are accumulated in variable $decrease_{i,j}$, with a more important weight put on the more frequent tasks (line 15). The program line $L_{i,j}$ with the largest weighted interference decrease is then selected (line 18 and 19). In practice, since the system may not necessarily be feasible (*e.g.* in the early stages of the algorithm, when almost nothing is locked in the cache), the iterative process used to compute the interference that T_i has on T_t does not necessarily converge (see § 2.3). We thus put a bound on the number of iterations (20 in the experiments) in order to *always* compute a response time, even when the system is not feasible with the current contents of the locked cache. When a program line $L_{i,j}$ is selected to be locked in the instruction cache, the WCET of the corresponding task T_i is updated (line 20).

Complexity. For every cache block, the algorithm scans all program lines $L_{i,j}$ that can be mapped into that block (at most η_s lines, with s the index of the set corresponding to the cache block). For every $L_{i,j}$, all tasks with a priority lower or equal than that of T_i (at most N tasks) are scanned in order to minimize the interferences caused by T_i on lower priority tasks. At most $B * max_{s\in[0..S-1]}(\eta_s) * N$ applications of CRTA are thus required, with B the number of cache blocks.

3.3 Identification of the worst-case execution sequences

Algorithms Lock-MU and Lock-MI are based on the knowledge of the worst-case execution path σ_i for every task T_i. Moreover, the optimality of Lock-MU with respect to the minimization of the CPU utilization is based on this knowledge. However, the worst-case execution path σ_i depends on the contents of the statically locked cache. At each step of the algorithms, in which a program line is selected to be locked in the cache, the worst-case execution path may change. To be optimal, the algorithms would have to solve a mutually recursive optimization problem, whose resolution would require to explore all the possible cache contents, which is not possible in practice. As a consequence, we chose to apply the algorithms to a single execution path only, which may not necesarily be the worst-case execution path for all possible cache contents. In the experiments detailed in section 4, the execution path used by the cache contents selection algorithms is obtained through an actual execution of the tasks with a cache disabled. This makes algorithm Lock-MU non optimal with respect to the minimization of the CPU utilization, but allows the algorithms to have a pseudo-polynomial complexity.

4 Performance evaluation

This section is devoted to an analysis of the worst-case performance of task sets using (i) the two static cache locking algorithms introduced in section 3, and (ii) a state of the art static cache analysis technique. The performance evaluation is preceded by a description of the experimental conditions under which the performance evaluation has been conducted.

4.1 Experimental setup

Target architecture and simulator. The target architecture considered in the experiments is the simplified MIPS processor used by the Nachos operating system. Nachos is a simple operating system for an emulated MIPS CPU, designed for educational purposes at the University of California at Berkeley. It has been extended for the purpose of this study with an optional instruction cache with blocks of size $S_b = 16$ bytes (4 MIPS instructions) and with a prefetch buffer of 16 bytes. The instruction cache can be parametrized so that its size S_c ranges from 512 bytes to 16 Kbytes, and so that its associativity degree W ranges from 1 (direct-mapped cache) to 32. The timing model considered for the processor is very simple: an instruction is assumed to execute in $t_{hit} = 1$ clock cycle in the case of a prefetch buffer/cache hit, and $t_{miss} = 10$ clock cycles otherwise. The execution sequences (σ_i) used by the cache contents selection algorithms are obtained through an execution of the tasks on Nachos with its instruction cache disabled.

Static WCET analyzer. The WCETs of tasks are computed using the Heptane static WCET analysis tool [CP01]. Heptane computes WCETs through a bottom-up traversal of the syntax tree of the analyzed C programs. Table 1 gives a flavor of the formulas used in Heptane to compute WCETs (these formulas are overly simplified since they assume constant and context-independent execution times for the leaves of the syntax tree).

	Formulas for WCET computation: $WCET(S)$
$S = S_1; ...; S_n$	$WCET(S) = WCET(S_i) + ... + WCET(S_n)$
$S =$ if (tst) then S_1 else S_2	$WCET(S) = WCET(Test) + \max(WCET(S_1), WCET(S_2))$
$S =$ loop$(tst)S_1$	$WCET(S) = maxiter * (WCET(Test) + WCET(S_1)) + WCET(Test)$ where $maxiter$ is the loop maximum number of iterations.
$S =$ branch-free sequence of instructions (basic block) BB_x	$WCET(S) =$ WCET of basic block BB_x

Table 1. Simplified equations for tree-based WCET computation

Task set "Small", 4 tasks, Total code size 6672 bytes				
Name	Description	Code size (bytes)	WCET (no cache)	Period
jfdctint	JPEG integer implementation of the forward DCT	3440	30773	85548
crc	CRC (Cyclic Redundancy Code) example program	1296	288769	551548
Integral	Integral computation by intervals	1152	175758	764547
matmult	Matrix multiplication	784	298247	1556849
Task set "Medium", 4 tasks, Total code size 13456 bytes				
qurt	Computation of roots of quadradic equations	1824	25434	70706
minver	Matrix inversion	4528	41550	79360
jfdctint	JPEG integer implementation of the forward DCT	3440	30773	133862
fft1	FFT (Fast fourier transform) Cooly-Turkey algorithm	3664	132486	691576

Table 2. Task sets characteristics

Heptane includes hardware modeling capabilities so as to estimate safely but precisely the WCETs on architectures with instruction caches, pipelines and simple branch predictors. Here, Heptane's pipeline and branch prediction modeling modules have been switched off since our focus is on instruction caches only.

The technique used in Heptane to estimate the worst-case behavior of applications with respect to the instruction cache is based on F. Mueller's so-called *static cache simulation* [Mue00]. The cache analysis technique computes *abstract cache states* (representation of all possible cache contents considering all possible execution paths in the program) using data-flow analysis on the program control flow graph. Abstract cache states are then used to classify the instructions according to their worst case behavior regarding the instruction cache (e.g. *hit* when it is certain that there are no conflict for a cache block, *miss* when no better classification can be found, and intermediate categories when there are conflicts inside loops [CP01]). In the case of set-associative caches, when a given instruction appears in different blocks of the same set of an abstract cache state, different categories can be obtained for the different instances of the instruction. In this situation, in order for the analysis to be safe, the instruction is classified with the most pessimistic category.

Heptane has been modified such that its cache analysis module can be replaced by a module that takes into account the presence of a locked cache. This new module classifies instructions into two categories: *miss* and *hit*. An instruction is classified as a *hit* if it is locked in the instruction cache or loaded in the prefetch buffer, and is classified as a *miss* otherwise.

Task sets. The performance of the two algorithms for cache contents selection presented in section 3 has been evaluated on two real (non synthetic) task sets, named hereafter "Small" and "Medium". Their characteristics are given in table 2. The table gives for every task: its name, a short description, the size of its code in bytes, its WCET in the case of a disabled instruction cache in cycles, and its period in cycles. The periods of tasks have been selected so that the CPU utilization computed with equation 1 equals 1.3 (i.e. the task set is not feasible if no instruction cache is used).

4.2 Worst-case performance analysis of locked cache and dynamic caches

Since the primary focus in hard real-time systems is to prove that all deadlines are met, we focus on the *worst-case* performance of both task sets. When cache locking is used, the cache-related preemption delay γ_i is constant and equal to the delay required to refill the processor prefetch buffer ($\gamma_i = t_{miss} - t_{hit}$). Otherwise, it is considered here that all cache blocks of a task have to be reloaded after a context switch ($\gamma_i = min(\frac{codesize(T_i)}{S_b}, B) * (t_{miss} - t_{hit})$).

The worst-case system performance of both tasks sets are given in Table 3. Each cell indicates whether the task set is feasible or not according to CRTA (equation 3). A shaded cell means that a task set is not schedulable. The

Asso \ Size		512B	1KB	2KB	4KB	8KB	16KB
1	Lock-MU	0.862	0.749	0.642	0.572	0.572	0.572
	Lock-MI	0.876	0.805	0.699	0.629	0.629	0.629
	Dynamic	0.608	0.488	0.397	0.406	0.406	0.406
2	Lock-MU	0.851	0.689	0.553	0.477	0.429	0.429
	Lock-MI	0.859	0.727	0.622	0.514	0.430	0.430
	Dynamic	0.740	0.562	0.488	0.406	0.406	0.406
4	Lock-MU	0.849	0.679	0.528	0.420	0.372	0.357
	Lock-MI	0.860	0.720	0.548	0.435	0.372	0.357
	Dynamic	0.943	0.741	0.567	0.498	0.406	0.406
8	Lock-MU	0.842	0.678	0.523	0.414	0.357	0.357
	Lock-MI	0.860	0.723	0.547	0.434	0.357	0.357
	Dynamic	1.162	0.968	0.747	0.576	0.498	0.406
16	Lock-MU	0.841	0.684	0.523	0.414	0.357	0.357
	Lock-MI	0.857	0.720	0.542	0.439	0.357	0.357
	Dynamic	1.276	1.187	0.971	0.757	0.576	0.498
32	Lock-MI	0.857	0.724	0.554	0.444	0.357	0.357
	Lock-MU	0.841	0.683	0.523	0.414	0.357	0.357
	Dynamic	1.306	1.283	1.190	0.981	0.757	0.576

(a) Task set "Small"

Asso \ Size		512B	1KB	2KB	4KB	8KB	16KB
1	Lock-MU	1.099	0.980	0.830	0.682	0.629	0.629
	Lock-MI	1.086	0.998	0.851	0.714	0.662	0.662
	Dynamic	0.744	0.578	0.493	0.502	0.505	0.505
2	Lock-MU	1.135	0.932	0.812	0.629	0.484	0.435
	Lock-MI	1.119	0.936	0.852	0.684	0.538	0.475
	Dynamic	0.798	0.683	0.585	0.510	0.505	0.505
4	Lock-MU	1.125	0.946	0.760	0.597	0.437	0.350
	Lock-MI	1.126	0.972	0.807	0.649	0.474	0.350
	Dynamic	0.982	0.810	0.714	0.616	0.516	0.505
8	Lock-MU	1.137	0.914	0.744	0.567	0.427	0.348
	Lock-MI	1.130	0.938	0.803	0.619	0.454	0.348
	Dynamic	1.101	1.014	0.843	0.740	0.622	0.516
16	Lock-MU	1.139	0.924	0.742	0.565	0.426	0.348
	Lock-MI	1.140	0.917	0.807	0.613	0.449	0.348
	Dynamic	1.162	1.165	1.077	0.865	0.744	0.622
32	Lock-MI	1.137	0.936	0.742	0.561	0.427	0.348
	Lock-MU	1.136	0.911	0.805	0.621	0.443	0.348
	Dynamic	1.293	1.232	1.244	1.103	0.868	0.744

(b) Task set "Medium"

Table 3. Compared worst-case performance of static cache locking and cache analysis

CPU utilization computed by equation 1 is also given in each cell. These two pieces of information are given for different cache sizes S_c, degrees of associativity W, and this with a dynamic cache and static cache analysis (line *Dynamic* in the table), and with static cache locking (lines *Lock-MU* and *Lock-MI* in the table).

Figure 3 depicts the CPU utilization obtained on task set "Small" from the contents of table 3(a). It compares the CPU utilization obtained when using cache locking (algorithm Lock-MI) and a dynamic cache together with static cache analysis. Very similar observations could be obtained when using algorithm Lock-MU.

Performance for small cache sizes and low degrees of associativity. It can be noted from the contents of figure 3 and table 3 that for small caches and low degrees of associativity, static cache analysis performs better than static cache locking, both from the standpoint of the task set feasibility and from the standpoint of CPU utilization. Actually, for small caches, there is a high degree of intra-task and inter-task conflicts for every cache block. The fact that static WCET analysis can take advantage of the spatial locality of the references to instructions results in lower WCETs than those obtained when cache locking is used. Even though the cache-related preemption delay is larger with dynamic caches compared to locked caches, its impact on the system schedulability and CPU utilization is negligible because of

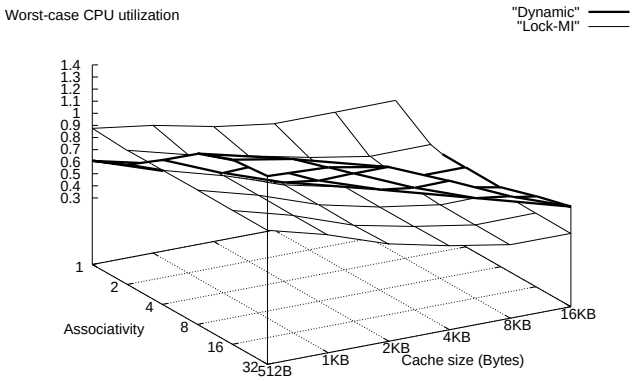

Figure 3. Worst-case CPU utilization of task set "Small"

the small cache size.

Impact of the cache size. For a given degree of associativity, the performance of both static cache locking and static cache analysis increases with the cache size, because of the decrease of the number of conflicts for cache blocks. However, the performance increase of static cache locking is higher than the one of static cache analysis when the cache size increases. For instance, for a 4-way set-associative cache, task set "Medium" exhibits better (i.e. lower) CPU utilization when using static cache locking than when using cache analysis for caches larger than 8 Kbytes, whereas static cache locking performs better for caches smaller than 8 Kbytes. This is because the cache-related preemption delay increases linearly with the cache size for the static cache analysis method, whereas it stays constant for the static cache locking method.

Impact of the degree of associativity. For a given cache size, the performance of static cache locking scales better than the one of static cache analysis with the degree of associativity W. Indeed, static cache locking takes benefit of the increasing degree of associativity to eliminate both intra-task and inter-task interference, which explains that the CPU utilization increases with W. In contrast, the static cache analysis method we have used does not scale well with W. This comes from the *pessimistic* way the instructions are classified (see § 4.1).

Compared performance of Lock-MU and Lock-MI. The first observation that can be made from the contents of table 3 is that the CPU utilization when using Lock-MI is in general worse (i.e. larger) than the CPU utilization when Lock-MU is used. However, there are some exceptions to this general rule (in particular for task set "Medium" and small caches). These exceptions come from the fact that an *actually measured* execution path is used in Lock-MU for contents selection instead of a *worst-case* path (see § 3.3). Remark that despite its utilization generally larger than the one of Lock-MU, algorithm Lock-MI accepts as many task sets as Lock-MU, and in some situations accepts tasks sets not feasible under Lock-MU (e.g. task set "Medium" for a 4-way set-associative cache of 1 Kbytes).

Tentative improvement of Lock-MI. In algorithm Lock-MI, each time a program line $L_{i,j}$ is selected to be locked in the cache, the WCET C_i of task T_i has to be updated (see figure 2, line 20). This update of C_i assumes that the selection of $L_{i,j}$ does not change the worst-case execution path, which may not be necessarily true (see section 3.3). Thus, we have slightly modified the algorithm so as to update C_i by resorting to a complete WCET analysis rather than approximating it. This modification consists in replacing line 20 in the algorithm given in figure 2 by C_i := *Estimate_WCET(Cache)*. This modification, while lowering the worst-case CPU utilization compared to the original Lock-MI did not change the schedulability of the task sets considered in the experiments, and considerably increased the computation time for cache contents selection, especially for large caches.

5 Concluding remarks

The key benefits of static cache locking is to make the time required to perform memory accesses predictable, and to be a unified technique to take into account both intra-task and inter-task conflicts for cache blocks. This class of techniques alleviates the need for using complex static analysis techniques to compute WCETs and cache-related preemption delays. In addition, it can be applied in situations where static cache analysis cannot be used at all (*e.g.* when the instruction cache has a non deterministic or non documented cache replacement policy). We have proposed in this paper two pseudo-polynomial algorithms for selecting the contents of statically locked caches. Although both algorithms have been introduced for strictly periodic tasks with deadlines equal to periods, they can be transposed in a straightforward manner to sporadic tasks or tasks with deadlines lower than periods. A worst-case performance analysis of the proposed algorithms on two task sets has shown that they perform better than static cache analysis for large instruction caches and caches with a large degree of associativity.

Concerning the performance of static cache locking, an average-case performance analysis is obviously required to assess the performance of static cache locking. Such a study has already been achieved for the task sets introduced in this paper, but its result is not included here for space considerations. A larger panel of task sets (including synthetic ones) is also required to explore the performance of the algorithms on task sets with more varied characteristics (code size, degrees of intra-task and inter-task interferences).

Concerning the algorithms for contents selection, an important technical issue concerning the proposed algorithms is the choice of the path according to which the selection of the cache contents is achieved. An analysis of the sensitivity of the algorithms regarding the choice of this path would be interesting. Future work is also required to extend them to data caches, unified caches and multi-level caches. The most challenging point is the extension of static cache locking to data caches, for which static analysis is particularly difficult due to fact that the addresses of most data structures are known only at run-time. We also plan to compare the performance of our algorithms with the ones of the genetic algorithms proposed in [CIBM01], both in terms of

algorithmic complexity and in terms of quality of the results produced (worst-case and average-case performance of the task sets). For large programs, we plan to explore more dynamic cache locking strategies. For instance, we plan to select different contents of the locked cache changed at statically-defined points in order to cope with the tasks dynamic behavior while staying predictable. Finally, an extensive study of actual hardware is required to assess the programming effort needed to lock values into the cache.

Acknowledgments

The authors wish to thank Jakob Engblom, Antonio Martí Campoy, Jean-Philippe Lesot and the anonymous reviewers for their valuable comments and observations.

References

[ABS01] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous memory management for embedded systems. In *Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Atlanta, GA, USA, November 2001.

[AFMW96] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer, September 1996.

[BMSO+96] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 1996 Real-Time technology and Applications Symposium*, pages 204–212. IEEE Computer Society Press, June 1996.

[BN94] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 1994.

[CIBM01] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask premptive real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, London, UK, December 2001.

[CP00] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, May 2000.

[CP01] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[HAM+99] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[KAS93] J. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, September 1993.

[Kir89] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS89)*, pages 229–237, Santa Monica, California, USA, December 1989.

[LHS+98] C. G. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6), June 1998.

[LL73] C.L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LMW96] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 254–263. IEEE, IEEE Computer Society Press, December 1996.

[LS99] Thomas Lundqvist and Per Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, November 1999.

[Mue00] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[PB00] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000. Guest Editorial.

[SS93] John E. Sasinowski and Jay K. Strosnider. A dynamic programming algorithm for cache/memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42(8):997–1001, August 1993.

[TBW94] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems*, 6(1):133–151, March 1994.