Dynamic Instruction Cache Locking in Hard Real-Time Systems

Alexis Arnaud Isabelle Puaut
IRISA, Campus universitaire de Beaulieu,
35042 Rennes Cédex, France
Email: {aarnaud/puaut}@irisa.fr

Abstract

Cache memories have been widely used in order to bridge the gap between high speed processors and relatively slower main memories, and thus to improve the overall performance of systems. However in the context of hard real-time systems, they are a source of predictability problems. A lot of progress has been achieved to model caches to statically determine safe and precise bounds on the worst-case execution times (WCETs) estimates of tasks on architectures with caches. Nonetheless cache-aware WCET analysis techniques may not always be applicable or may be too pessimistic, because some memory accesses are unknown statically. Another reason may come from a poorly documented or non-deterministic cache line replacement policy. An alternative approach is to lock cache lines so as to make memory access times entirely predictable.

In this paper, we consider an instruction cache and a task. We propose a an algorithm which partitions the task into a set of regions. Each region owns statically a locked cache contents determined offline.

A set of tasks is used to experimentally analyze the effects of the algorithm on the worst-case cache miss rate (WCCMR). A sharp improvement is observed, as compared with a system without any cache. Furthermore it is observed that the results obtained on WCCMRs compare to the results obtained from static analysis of a cache whose policy is to replace least recently used (LRU) cache lines. Contrary to cache analysis techniques, our algorithm depends neither on the scheduling policy, nor on the cache line replacement policy. As a further property, it works at the machine language level, and thus does not require any source code.

Keywords : hard real-time systems, cache memories, worst-case execution time

1 Introduction

1.1 Cache memories and real-time issues

Caches are small buffer memories with low latency which are inserted between the CPU and the main memory. They benefit from the spatial and temporal locality often found in instruction and/or data streams in order to store, at any time, memory references which are likely to be addressed in a near future. They operate transparently. Therefore no change is required in the memory addressing scheme. They bring an improvement of the overall performance of computer systems. However two phenomena make it hard to know statically memory access in the worst case:

- Intra-task interferences which occur when a task overrides its own cache lines, mainly because of the relatively small size of the cache as compared with the task's memory demands.
- In preemptive multitasking systems, preemptions cause inter-task interferences. Namely when the execution is switched from a task A towards a task B, some cache blocks used by A may be evicted by B.

In the industry, there is a growing demand of hard real-time systems with improved performance and cheaper hardware. Thus the challenge here is to accomodate the performance goal of cache memories with predictability requirements of hard real-time systems.

1.2 Cache memories in hard real-time systems

There are at the present time two categories of approaches for safely incorporating cache memories in hard real-time systems. In the first one, *cache analysis*, caches operate without any restriction. Static analysis techniques (cache-aware WCET analysis [9, 7] and schedulability analysis

[6]) predict their worst-case impact on the system schedulability. They assume that the cache line replacement policy is known.

The second category of approaches consists in using caches in a restricted or customized manner in order to adapt them to the needs of hard real-time systems and schedulability analysis.

Cache-partitioning techniques assign portions of a cache to some specified tasks in order the guarantee that for each task its most recently used code or data will remain in the cache while the processor executes another task. The partitioning can be made at the hardware [5] or software level [8]. Since the dynamic behavior of the cache is isolated within each partition, inter-task interferences are eliminated. The counterpart is that the per-task available amount of memory is reduced, hence decreased performance. Furthermore static cache analysis is still required to tackle intra-task interferences.

An alternative is to use *cache locking techniques*. Locking a cache line consists in loading some contents in a cache and inhibiting the cache line replacement policy. If all the cache lines are locked, we say that the state of the cache is a *locked cache state*. Predictability is strictly ensured if contents is chosen offline. This feature is available on several commercial processors (among others: Motorola ColdFire MCF5249, Motorola PowerPC 603e, ARM 940T).

Given a task, its code is subdivided into one or more zones. Each such zone has a locked cache state. Consequently, executions of the task are subdivided into temporal windows, in each of which the cache is locked. When there is more than one zone, the locking scheme is said to be *dynamic*, whereas for only one zone, it is *static* [11, 2].

If the locking method is *global*, at every instant, each task owns a portion of the cache space. No cache reload is needed when a task is preempted. In the case of a *local* locking method (see for example [10, 3]), each task owns the entire cache. To ensure this, the cache is reloaded each time a preemption occurs.

1.3 Paper contents and contributions

This paper explores the use of *local dynamic locking* of instruction caches in hard real-time systems. Dynamic cache locking is attractive from several points of views. First of all, it improves the worst and average-case performance of tasks, as compared with the case where the same tasks do not use any cache at all.

When using dynamic instruction cache locking techniques, the interactions between the dynamic properties of caches and other architectural components such as pipelines or branch predictors are less complex, making easier the analysis of these components in validation tools of hard real-time systems. Dynamic instruction cache locking can also be used when no cache analysis method can apply accurately, due for instance to non-deterministic or poorly documented cache replacement strategies.

It may be also suitable for designing mixed systems providing both tasks with hard real-time constraints and tasks with soft real-time constraints which may use unrestricted caches.

In this paper, we propose algorithms for finding a partition of the machine code of a given task into regions, and to determine a locked state of the instruction cache for each such region. It is performed in a non-blind manner by using memory access patterns obtained by profiling the task. The goal is to improve the worst-case performance as compared with a system with no cache, in such a way that this performance be comparable with results obtained from static analysis of the same cache whose replacement policy is the least recently used (LRU).

1.4 Paper organization

The remainder of the paper is organized as follows. Section 2 gives an overview of the proposed local dynamic instruction cache locking strategy. Then we detail the experimental setup and performance measurements used for validating our approach in Section 3. In Section 4, we give an overview of other studies related to our work. Finally we conclude in Section 5 with a summary of the paper contributions.

2 A dynamic instruction cache locking technique

In this section we describe our method which supports dynamic instruction cache locking. After introducing the assumptions and notations (§2.1) and giving a first glance at the method (§2.2), the central objects of this work, namely regions, are studied in paragraph 2.3. Then we detail how to associate a locked cache state to a region of a program in order to improve the worst-case performance of this program (§2.4). Finally, an algorithm for partitioning a program into such regions is described in the paragraph 2.5.

2.1 Assumptions

2.1.1 Architecture and program model

In our model, we consider a CPU provided with a one-level set-associative instruction cache.

We will consider a progamm presented in binary form. Each subroutine owns a unique return point. Indirect jumps are excluded. Moreover the program is assumed to execute within a finite amount of time.

Throughout this paper, for any program, we will associate to each of its subroutines a control flow graph (CFG). A control flow graph is an abstract representation of a subroutine. Each node in the graph represents a basic block, i.e. a sequential piece of code with a unique entry point and a unique exit point. Directed edges are used to represent jumps in the control flow.

2.1.2 Reloading and locking operation

Reloading and locking the cache may be done by inserting instructions calling a special subroutine. However, in this work, this operation is assumed to be done without modifying the program. We use debug registers which raise an exception at specified values of the program counter. An exception handler does the job of reloading and locking the cache. The benefit is that the program's memory map is left unchanged.

2.2 Overview

We propose to apply a local dynamic cache locking strategy which aims to improve the WCET of a program as compared with the case of a system with no cache. The main issue is to avoid performing an exhaustive search of all the possible subdivisions of the program and of all possible cache contents for each subdivision, as this would result in a combinatorial explosion. The proposed method consists in the following two steps :

1. **Profiling.**
   We determine, from executions of the program with various entry data sets, a collection of execution paths along with their execution frequencies. These paths must verify the following two conditions: (i) as many basic blocks as possible are reached; (ii) no path can be deduced from other paths with set operations, so that the number of paths is minimal. From this profiling information, we compute for each basic block an execution frequency.

2. **Program partitioning**
   A greedy algorithm is applied on the set of basic blocks. At the initial state, the program is presented as the set of basic blocks of its control flow graph. Each such basic block is a *region*. At each step of the algorithm, regions are aggregated into new regions. Each region has a locked cache state. Two basic operations, merging and inlining, allow to create new regions from existing ones. The goal of the algorithm is to determine a set of regions minimizing the WCET estimation of the program.

2.3 Regions

The notion of region is central in this work. Given a subroutine whose CFG is known, a region R is a connected part of this CFG. Namely, between any couple B_1, B_2 of basic blocks of R, there exists at least one non-directed path between them. R may be of one of two types :

- R is a *simple region* if it has a locked cache state which is known statically. This state is computed with an algorithm described in section 2.4. The addresses through which other regions of the program may enter R are *cache reload points*. When one of them is reached, the cache is reloaded with the locked cache state of R.

- Suppose R spans all the basic blocks of its subroutine. If there is a significant benefit from avoiding cache reloads when entering and exiting from this subroutine, R may be *inlined*. In this case, R *inherits* the cache state of any region in which the subroutine was called.

2.4 Computation of a locked cache state for a simple region

Consider a simple region R in a program. We provide it with a locked cache state. Namely, for each cache line, we select from this region the memory line such that: (i) it can be loaded in that cache line; (ii) its execution frequency is the highest; (iii) the gain obtained from having this memory line in the cache is more important than the cost of loading and locking it in the cache.

The last condition is true if the execution frequency of this memory line exceeds a constant proportional to the average number of times a cache reload occurs when entering R.

This locked cache state is chosen so as to minimize, among all possible choices, an heuristic which is the approximate time spent, during any execution of the program, in the basic blocks of R plus the average time spent reloading the locked cache state of R. The proof of this property vaguely follows the lines of the main proof presented in [11], so we will not detail it here.

2.5 The Region Merging and Inlining algorithm

In this Section, first we define two basic operations on regions, namely merging and inlining (§2.5.1). Then, in order to improve the WCET of a program, an algorithm (§2.5.2) partitions the program into regions using these two operations.

2.5.1 Basic operations on regions

Merging Let R_1 and R_2 be two simple regions that are connected. Merging these two regions into a new simple region R means that:

- R aggregates the blocks of R_1 and R_2
- The locked cache state of R is computed by the algorithm presented in the paragraph 2.4.

We will use the notation $R = R_1 \oplus R_2$ to express the fact that the region R is obtained as the result of merging R_1 with R_2.

Inlining Suppose a subroutine contains only a simple region R. There may be a potential benefit by avoiding cache reloads when calling and exiting this subroutine. The general idea for the inlining operation is to allow the this subroutine to inherit the locked cache state from the subroutine which has just called it.

We now define a *calling region* CR of R the following way (cf. figure 1):

- CR is a simple region.
- There exists at least one chain $(f_0, \ldots, f_{m-1})$ of subroutine calls leading from CR to R: (i) the call towards f_0 lies in CR; (ii) if $m \geq 2$, each $f_0, \ldots f_{m-2}$ represent an inlined region; (iii) f_{m-1} calls towards the subroutine representing R.

Now let CR_i $(1 \leq i \leq n)$ be the calling regions of R. Then *inlining* R means that, for each CR_i:

- For each memory line of R, its frequency is assumed to be scaled up by the proportion, among all the calling regions, of calls from CR_i towards R.
- Its locked cache state is computed (§2.4) offline from the knowledge of the memory lines of both CR_i and R.

From now on, the locked cache state of R is inherited from the locked cache state of the last accessed calling region during runtime (cf. figure 1).

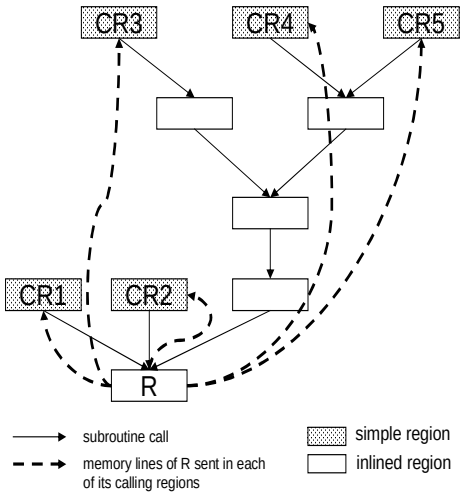

Figure 1. A region R and its calling regions CR_i. Inlining operation on R.

2.5.2 Description of the algorithm

In this subsection, we give a description of an algorithm which determines a partition of a program P into regions in order to to minimize the WCET of P. We propose a sub-optimal strategy, the RMI (Region Merging and Inlining) greedy algorithm. RMI takes as an input the partition of P in basic blocks, which are initial regions. At each iteration, a pair of regions is chosen and merged once for all, thus giving a new partition choice. RMI keeps also track of the current best partition. Inlining operations are involved when updating the best partition. When completed, RMI returns the best found partition of the program.

Quality of an operation. Let Ω_{pre} be a partition of P, and Ω_{post} the partition resulting from an operation (merging or inlining) on Ω_{pre}. The quality criterion of this operation is based on the difference, noted δ, between the WCETs of P with the locked cache states from respectively Ω_{post} and Ω_{pre}. The best operation gives the lowest value of δ, noted δ_{min}. It represents on the WCET of P its best improvement if $\delta_{\text{min}} < 0$, and its least deterioration otherwise.

In order to choose among some possible operations on Ω_{pre} the best one, the EvalOp algorithm (cf. algorithm 1) must be called each time such an operation was attempted on Ω_{pre}. Given an operation, the WCET of the resulting partition Ω_{post} and its quality criterion δ are computed (ℓ. 1-2). Then EvalOp updates the information on the partition resulting from the best operation on Ω_{pre} if needed (ℓ. 3).

Algorithm 1 EvalOp algorithm

Require: P: program; Ω_{pre}: partition of P; Ω_{post}: partition after an operation; δ_{min}: best quality criterion
Ensure: Ω_{min}: partition resulting from the best operation; δ_{min}
1: WCET(Ω_{post}) $\leftarrow$ WCET of P with Ω_{post};
2: $\delta \leftarrow$ WCET(Ω_{post}) - WCET(Ω_{pre});
3: **if** $\delta \leq \delta_{\text{min}}$ **then** $\Omega_{\text{min}} \leftarrow \Omega_{\text{post}}$; $\delta_{\text{min}} \leftarrow \delta$;

Description of the RMI algorithm. First note that a partition Ω of P into regions gives rise to a search space. Namely this search space contains all the partitions that can be deduced from Ω by operations (merging and inlining) on its regions.

Algorithm 2 RMI algorithm

Require: P: program, Ω_{init}: initial partition of P, S_{max}: max size of a set of locked cache states
Ensure: Ω_{best}: best found partition
1: $\Omega_{\text{cur}} \leftarrow \Omega_{\text{init}}$; $\Omega_{\text{best}} \leftarrow \emptyset$;
2: WCET(Ω_{cur}) $\leftarrow$ WCET of P with Ω_{cur};
3: WCET(Ω_{best}) $\leftarrow$ WCET of P (no cache);
4: **while** a subroutine has more than 1 simple region in Ω_{cur} **do**
5: $\quad\Omega_{\text{cur}} \leftarrow$ TryMerge(P, Ω_{cur});
6: $\quad$**if** WCET(Ω_{cur}) $\leq$ WCET(Ω_{best}) and Size(Ω_{cur}) $\leq S_{\text{max}}$ **then** $\Omega_{\text{best}} \leftarrow \Omega_{\text{cur}}$;
7: $\quad\Omega_{\text{inlined}} \leftarrow \Omega_{\text{cur}}$;
8: $\quad$**while** there are inlineable regions in Ω_{inlined} **do**
9: $\quad\quad\Omega_{\text{inlined}} \leftarrow$ TryInline(P, Ω_{inlined});
10: $\quad\quad$**if** WCET(Ω_{inlined}) $\leq$ WCET(Ω_{best}) and Size(Ω_{cur}) $\leq S_{\text{max}}$ **then** $\Omega_{\text{best}} \leftarrow \Omega_{\text{inlined}}$;
11: $\quad$**end while**
12: **end while**

The RMI algorithm (cf. algorithm 2) starts from the initial solution search space corresponding to the basic blocks of P stored in the current partition choice Ω_{cur} (ℓ. 1). At each iteration, RMI searches for the best merging between a pair of regions (ℓ. 5) by calling the TryMerge algorithm (cf. algorithm 3), thus updating Ω_{cur}, and equivalently reducing the solution search space. It then updates the information on the best partition (ℓ. 6-11). The whole process is iterated until no merging operation is possible in the solution search space, which means that, in Ω_{cur}, only one simple region remains in each subroutine (ℓ. 5). When choosing the best partition Ω_{best} of P, RMI first compares Ω_{best} against Ω_{cur} (ℓ. 6), and updates it if needed. Then, starting from Ω_{cur}, a greedy algorithm is used to choose a sequence of inlining operations (ℓ. 8-11) by calling the TryInline algorithm (cf. algorithm 4). At each step, the current choice is stored in Ω_{inlined}. After a choice was made, Ω_{best} is updated if needed.

Description of the TryMerge algorithm. Given a partition Ω of the program P, for each pair of mergeable regions, the TryMerge algorithm tries to merge them and builds a test partition Ω_{test} (ℓ. 3). If the EvalOp algorithm decides that Ω_{test} results from the current best merging operation, it is stored in Ω_{min} (ℓ. 4). After completion, Ω is updated with the partition stored in Ω_{min} representing the best merging operation (ℓ. 6).

Algorithm 3 TryMerge algorithm

Require: P: program, Ω: partition of P
Ensure: Ω
1: $\delta_{\text{min}} \leftarrow \delta_{\text{max}}$;
2: **for** each connected pair of simple regions $(R_1, R_2) \in \Omega$ **do**
3: $\quad\Omega_{\text{test}} \leftarrow (\Omega \backslash \{R_1, R_2\}) \cup \{R_1 \oplus R_2\}$;
4: $\quad(\Omega_{\text{min}}, \delta_{\text{min}}) \leftarrow$ EvalOp(P, Ω, Ω_{test}, δ_{min});
5: **end for**
6: $\Omega \leftarrow \Omega_{\text{min}}$;

Description of the TryInline algorithm. Given a partition Ω of P, for each subroutine which contains only one simple region R, the TryInline algorithm builds a test partition Ω_{test} in which R is inlined (ℓ. 3-7). As for TryMerge, the EvalOp algorithm is used to choose the current best inlining operation (ℓ. 8) whose corresponding partition is stored in Ω_{min}. After completion, Ω contains the partition corresponding to the best inlining operation (ℓ. 10).

Algorithm 4 TryInline algorithm

Require: P: program, Ω: partition of P
Ensure: Ω
1: $\delta_{\text{min}} \leftarrow \delta_{\text{max}}$;
2: **for** each inlineable region $R \in \Omega$ **do**
3: $\quad\mathcal{CR}$: set of calling regions of R in Ω;
4: $\quad R' \leftarrow R$; $\mathcal{CR}' \leftarrow \mathcal{CR}$;
5: $\quad\Omega_{\text{test}} \leftarrow \Omega \backslash \{R, \mathcal{CR}\}$;
6: $\quad$Inline R' in $\mathcal{CR}'$;
7: $\quad\Omega_{\text{test}} \leftarrow \Omega_{\text{test}} \cup \{R', \mathcal{CR}'\}$;
8: $\quad(\Omega_{\text{min}}, \delta_{\text{min}}) \leftarrow$ EvalOp(P, Ω, Ω_{test}, δ_{min});
9: **end for**
10: $\Omega \leftarrow \Omega_{\text{min}}$;

As regards the worst-case complexity of the RMI algorithm in terms of the basic operations involved, merging and inlining, it is quadratic in the

number of basic blocks of the considered program. This property is shown in the annex A.

3 Experimental results

This section deals with an experimentation designed to validate the approach adopted in this work. In the paragraph §3.1, the experimental protocol and the assumptions are detailed. Then in the following paragraph (§3.2), we evaluate the impact of our method on the worst-case performance.

3.1 Experimental setup

Hardware and timing model. As the worst-case performance with regards to an instruction cache is our only concern, we assume an executive support from a 32 bit MIPS R3000 processor at instruction level only. In our model, this processor provides only one architectural component, namely an instruction cache. Its cache line replacement policy is the LRU policy. Moreover we suppose that this cache can be totally locked.

The size of the cache ranges from 512 bytes to 4 kilobytes, and its associativity is equal to 1 (thus it is direct-mapped). The application performance with respect to the cache is our only concern in this study. Therefore the timing model for the processor is very simple. The worst case performance of a task under a given configuration of the cache is measured in worst case cache miss rate (WCCMR).

When the cache is dynamically locked, a special routine of the underlying operating system is assumed to manage the reloading and the locking of the cache. *As the performance of this routine is highly critical, it is assumed to be stored into a scratch-pad memory* [12]. As we focus on cache misses, only operations loading memory lines into the cache are taken into account. Thus, given a locked cache state S, the worst number of cache misses of this routine is assumed to be equal to $|S|$, i.e. the number of cache lines in S.

Generation of execution traces. In order to profile programs, a MIPS R3000 processor emulator at instruction level is used to generate execution traces.

Estimation of worst case miss rates. The WCCMRs of programs, presented in binary form, are computed with the Heptane [1] static WCET analysis tool [4]. Within the context of this work, it uses a technique based on abstract syntaxic trees. In such a tree, the leaves are basic blocks, while the other nodes are sequences, if-then-else control structures, or loop structures. The WCET and the WCCMR are computed bottom-up by formulae which establish for each node a partial WCET (resp. WCCMR) depending on its children nodes. The WCET (resp. WCCMR) of the root node is then the WCET (resp. WCCMR) of the analyzed program.

Heptane includes hardware modeling capabilities to estimate safely but precisely the numbers of hits and misses in the worst case on architectures with instruction caches, pipelines and simple branch predictors. In the present study, Heptane's pipeline and branch prediction modeling modules were switched off since our focus is on instruction caches only. In addition of a cache analysis module, Heptane was incorporated a module that takes into account the presence of a dynamically locked instruction cache. This new module uses a file describing the set of cache states and cache reload points of the program to be analyzed. It classifies instructions into two categories : *miss* and *hit*. An instruction is classified a a *hit* if it is locked in the instruction cache, and is classified as a *miss* otherwise.

Experimentation process. Given a program and a parametrization of the instruction cache, the experiment proceeds in two steps (see figure 2). First, the set of cache states and cache reload points is computed by the RMI algorithm. For this purpose, execution traces are generated with Nachos.
The second step is the performance evaluation itself. The WCCMR is computed with Heptane. Two cases are considered: (i) a system with a dynamic instruction cache (i.e. operating in its normal behavior); (ii) a system with a dynamically locked instruction cache.

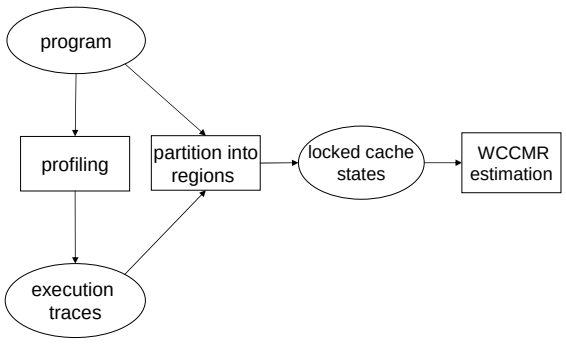

Figure 2. Experimental protocol

The experiments were conducted on three benchmark tasks, whose features are summarized

[1] Heptane is an open-source software available at http://www.irisa.fr/aces/software/software.html

in figure 3. The third column gives, for each task, the code size in bytes.

Name	Description	Size
minver	matrix inversion	4584
matmult	matrix product	1328
jfdctint	integer DCT transformation	3424

Figure 3. Characteristics of tasks

3.2 Performance of dynamic instruction cache locking

In this paragraph, we interpret the results obtained from the experimentation. First, we examine the worst-case performance improvement obtained with our approach (§3.2.1). Then we study some properties of the RMI algorithm itself (§3.2.2).

3.2.1 Worst-case performance

We compare the worst-case performance of the tasks in two situations: (i) the cache is dynamically locked; (ii) the cache is dynamic wih a LRU policy. The figures 4, 5, and 6 describe the results of the experiments. In the locked case, the WCCMR comprises the cache misses due to the task itself, and the cache misses arising from cache reloading operations.

Impact of the cache size. As seen on figures 4, 5, and 6, in both locked and LRU cases, the worst-case performance is far better than without any cache (in this situation, the WCCMR would be equal to 100%).

In the dynamic case, the WCCMR sharply decreases when increasing the cache size, as the cache conflict probability decreases.

In the locked case, when increasing the cache size, we observe a general tendency towards the decrease of the part of the WCCMR which represents the reload overhead, . But for a notable exception in the case of the task jfdctint with a 1 KB cache, a similar tendency applies as regards the part of the WCCMR from the task itself.

Now we compare the worst-case performance between the locked cache and the dynamic case. In this aim, we compute, for each task and each cache size, a ratio between the total WCCMR in the locked case and the WCCMR in the dynamic case. With the exception of two results (jfdctint with a 1 KB cache, and minver with a 4 KB cache), the average ratio is equal to 1 for jfdctint, 1.44 for minver and 0.83 for matmult. Thus the results are in the same order of magnitude in the locked and dynamic situations.

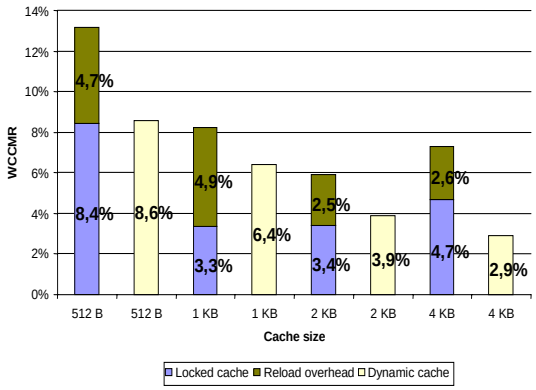

Figure 4. WCCMR results for minver.

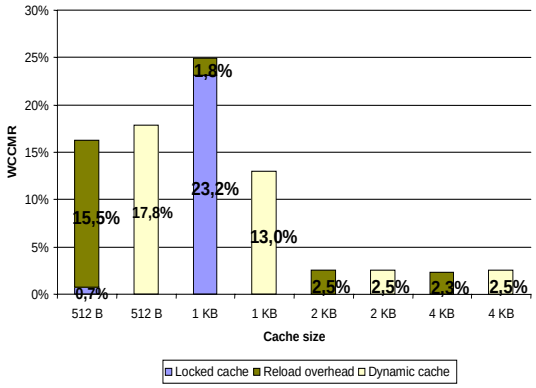

Figure 5. WCCMR results for jdctint.

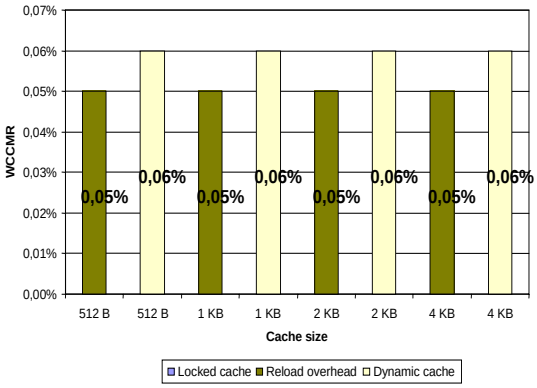

Figure 6. WCCMR results for matmult.

Impact of the associativity degree. On the figure 7, we consider the task minver and a 1 KB cache. The impact of the associativity degree is illustrated both in the locked and LRU cases.

We observe that the worst-case performance of the locked cache scales well when increasing the associativity degree. This can be explained by the fact that, for a given cache size, a cache contents computed by the Lock-MP algorithm for a specified associativity degree remains valid for other associativity degrees. This confers to the RMI algorithm a low sensitivity to the variations of this parameter.

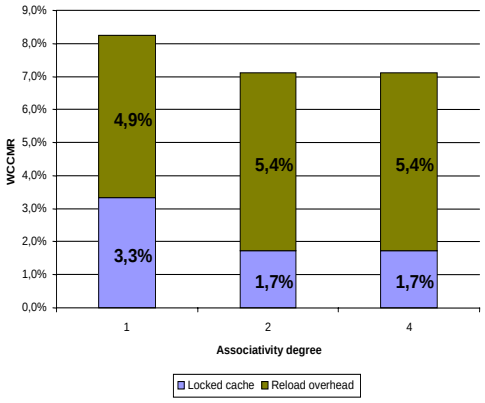

Figure 7. Compared impact of the associativity degree of a 1 KB cache for the task minver.

3.2.2 Properties of the RMI algorithm

Performance. As noticed before, the worst-case complexity of the RMI is quadratic in the number of basic operations, merging or inlining, on the initial set of basic blocks of a task. The figure 8 indicates, among others, for each task and for each cache parametrization, the number IB of initial basic blocks and the time T it took to compute a set of locked cache states. The study of the quantity T/IB^2 shows that the computation time T in seconds follows the approximate law $T = 0.5IB^2$.

Locked cache states. As regards the number of locked cache states determined by the RMI algorithm, the figure 8 shows that this number decreases when the size of the cache increases. This is is essentially due to the fact that the cache contents selection algorithm Lock-MP accepts a more important number of useful memory lines in a less important number of locked cache states.

A notable fact is that, even when the size of a task is inferior or approximately equal to the size of the cache, the RMI algorithm may determine more than one locked cache state. It can be seen in the figure 8 in the case of the task minver for a 4 KB cache, and in the case of the task matmult for most of the cache sizes. A reason for rejection of valuable memory lines by the Lock-MP algorithm is the existence of conflicts due to placement constraints in a set-associative cache. The RMI algorithm may address this issue by creating more locked cache states when there is a benefit from considering those rejected memory lines.

Task	Cache size	Nb of basic blocks	Computation time	Nb of cache states
minver	512 B	135	3h 6min 49s	10
	1 KB	135	2h 30min 52s	6
	2 KB	135	2h 35min 13s	4
	4 KB	135	2h 42min 12s	3
jfdctint	512 B	23	5min 22s	12
	1 KB	21	4min 32s	3
	2 KB	19	3min 12s	3
	4 KB	19	3min 13s	1
matmult	512 B	23	2min 48s	2
	1 KB	23	2min 48s	2
	2 KB	22	2min 50s	2
	4 KB	23	3min	2

Figure 8. Results characteristics

4 Related work

Studies have been performed for static instruction cache locking in multitasking hard real-time systems. In [2], a global approach is proposed. The cache state minimizing the cache-aware response time (CRTA) [1] of each task is chosen. It is achieved with a genetic algorithm. The fitness function is a weighted mean of the response time of each task. The same authors explored a local approach in [3] with the same algorithm for cache contents selection.

In [11], two greedy algorithms have been designed for a global locking scheme. Both have a pseudo-polynomial complexity. From task periods and access statistics of instruction blocks along the worst-case execution path of each task, each algorithm selects a cache state so as to minimize a well chosen cache-aware metric, and thus to improve the task set schedulability. A local variant is proposed in [10].

As explained in [10], static cache locking lacks some scalability. If the ratio between the size of the task set and the size of the cache memory is very high, only a very small fraction of the task set will benefit from the cache. Our work is applied on a per-task basis, and thus is a local approach. It is designed to overcome the scalability problem by al-

lowing the locked state of the cache to be reloaded at some addresses of a program.

The work [13] is a combination of dynamic data cache locking and static cache analysis. Given a task, at compile time, an algorithm computes the regions in the code where one cannot accurately determine all possible cache contents required for analyzing the state of the data cache, because of memory references which cannot be statically known. Such regions are enclosed with a pair of statements so that the cache is locked in them. A locality analysis based on the study of reuse vectors selects the data to be loaded in the cache. In order to address the multitasking issues, it is assumed that the data cache is partitioned among the tasks of the system. Also the knowledge of the cache replacement policy is required.

As compared with this work, our approach proposes a scheme in which the instruction cache is *always* locked. Thus our method does not depend on the cache line replacement policy, and may be used in cases when static cache analysis fails. Moreover our work does not depend on any partition of the cache. Therefore it does not require additionnal partitioning techniques, and it can be easily applied in situations in which the number of tasks of the system may vary.

Finally, scratch-pad memories [12] are an alternative to instruction or data caches. These are on-chip static memories with low latencies. As a consequence they may reconcile performance and predictability. They generally provide lower capacities than caches and consume far less power. Because of the addressing scheme, the code of tasks must be explicitly modified in order to benefit from scratch-pad memories. Thus, as compared with our scheme, this approach requires more compiler support. We believe that the addressing transparency provided by instruction caches is a key advantage, because it alleviates the need for code transformations.

5 Conclusion

The key benefit of instruction cache locking is to make the memory access times entirely predictable and to be a technique that eliminates intra-task conflicts. It can be applied in situations where static cache analysis cannot be used (e.g. when the cache has a non deterministic or undocumented cache line replacement policy). Moreover, it may make easier the analysis of other architectural components. In this work, we have proposed a local dynamic cache locking strategy and an algorithm for determining a finite number of cache configurations for a given task. Its additional features are independence from any scheduling policy (it is a per-task strategy), unnecessity to access the source code of programs, scalability with regards to cache associativity. With regard to performance evaluation against a system without any instruction cache, a sharp improvement is observed on the miss rates in the worst case. Moreover for many cache parametrizations, the worst-case performance is in the same order of magnitude as results from static LRU cache analysis.

As a further work, it would be interesting to explore the transposition of the RMI algorithm (i.e. we keep the basic merging and inlining operations) from a greedy algorithm towards a genetic algorithm. The main reason is that a genetic algorithm exhibits a better exploration of a solution space and thus might find sets of locked cache states which would lead to better improvements on worst-case performances. Another direction would be to adapt this work in other situations. It could be easily achieved for multi-level instruction caches. Finally, the adaptation to data caches should be investigated.

A Worst-case complexity of the RMI algorithm

As regards the worst-case complexity of the RMI algorithm, we now show that it is quadratic in terms of involved operations (merging and inlining). First we detail a worst-case scenario. Suppose our program comprises N_S subroutines $F_0 \dots F_N$, each with N_R basic blocks assimilated to simple regions. In each F_k, the N_R regions are consecutive. We consider the following calling hierarchy: for each k, F_k calls the subroutines $F_{k+1} \dots F_{N_S-1}$. For the sake of simplicity, we assume here that the main subroutine may be inlined. Starting from the value $k = 0$, RMI repeats the following steps until only one simple region remains in each subroutine: (i) choose a pair of regions of F_k, then merge them; (ii) in the remaining subroutines $F_{k+1}, \dots, F_{N_S-1}$, no pair of regions is chosen for merging; (iii) if only one simple region remains in F_k, then increment k; (iv) try to inline each of the subroutines $F_0, \dots, F_{k-1}$, but never choose one.

Given a value of k, each of the subroutines $F_0, \dots, F_{k-1}$ contain only one region simple. At a given stage, assume the subroutine F_k has $N_R - i + 1$ regions. Then $N_R - i$ mergings are tried before making a choice. As each of the remaining $N_S - k - 1$ subroutines $F_{k+1}, \dots, F_{N_S-1}$ has N_R simple regions, overall $(N_S - k - 1)(N_R - 1)$

mergings are tried without any choice being made. As regards the subroutines $F_0, \ldots, F_{k-1}$, k inlining operations are tried without any success. Thus, at a given stage, $(N_R - i) + (N_S - k - 1)(N_R - 1)$ operations are done. As the number of regions of F_k can vary from 2 to N_R for mergings, i ranges from 1 to $N_R - 1$. Now summing over the N_S subroutines, we obtain the following number of operations: $\sum_{k=0}^{N_S-1} \sum_{i=1}^{N_R-1} [(N_R - i) + (N_S - k - 1)(N_R - 1) + k]$. The computation of this sum yields $\frac{1}{2} N_S^2 N_R (N_R - 1)$ operations. Thus this value is in $O((N_S N_R)^2)$. As $N_S N_R$ is the number of basic blocks of the program, the worst-case complexity of the RMI algorithm in number of operations is quadratic with the number of basic blocks.

References

[1] 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 (RTAS '96)*, pages 204–212. IEEE Computer Society, June 1996.

[2] A. Marti Campoy, A. P. Ivars, and J. V. Busquets Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of the IEEE/IEE Real-Time Embedded Systems Workshop (RTES'01)*, 2001.

[3] A. Marti Campoy, A. P. Ivars, F. Rodriguez, and J. V. Busquets Mataix. Static use of locking caches vs. dynamic use of locking caches for real-time systems. In *IEEE Canadian Conference on Electrical and Computer Engineering*, Montreal, Canada, May 2003.

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

[5] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS '89)*, pages 229–237, Santa Monica, CA, USA, December 1989.

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

[7] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, 1999.

[8] F. Mueller. Compiler support for software-based cache partitioning. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 125–133, New York, NY, USA, 1995. ACM Press.

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

[10] I. Puaut, A. Arnaud, and D. Decotigny. Analyse de performance de méthodes de verrouillage statique de caches dans les systèmes temps-réel strict. In *Proc. of the 12th International Conference on Real-Time Systems (RTS'04)*, March 2004.

[11] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE International Real-Time Systems Symposium (RTSS '02)*, Austin, TX, USA, December 2002.

[12] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 213–218, New York, NY, USA, 2002. ACM Press.

[13] X.Vera, B. Lisper, and J.Xue. Data caches in multitasking hard real-time systems. In *24th IEEE International Real-Time Systems Symposium (RTSS '03)*, Cancun, Mexico, 2003. IEEE Computer Society Press.