WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems

Isabelle Puaut

Université de Rennes I/IRISA
Campus Universitaire de Beaulieu, 35042 RENNES Cedex - France

Abstract

Cache memories have been extensively used to bridge the gap between high speed processors and relatively slower main memories. However, they are sources 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 worst-case execution times (WCETs) of tasks on architectures with caches. However, cache-aware WCET analysis techniques are not always applicable due to the lack of documentation of hardware manuals concerning the cache replacement policies. Moreover, they tend to be pessimistic with some cache replacement policies (e.g. random replacement policies) [6]. Lastly, caches are sources of timing anomalies *in dynamically scheduled processors [13] (a cache miss may in some cases result in a shorter execution time than a hit).*

To reconciliate performance and predictability of caches, we propose in this paper algorithms for software control of instruction caches. The proposed algorithms statically divide the code of tasks into regions, for which the cache contents is statically selected. At run-time, at every transition between regions, the cache contents computed off-line is loaded into the cache and the cache replacement policy is disabled (the cache is locked*). Experimental results provided in the paper show that with an appropriate selection of regions and cache contents, the worst-case performance of applications with locked instruction caches is competitive with the worst-case performance of unlocked caches.*

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 rely on the knowledge of an upper bound for the task execution times (WCETs, for Worst-Case Execution Times). WCET estimates have to be *safe* (i.e. greater than any possible execution time) and as *tight as possible* (as close as possible as the execution time of the longest path). Safe bounds for task execution times can be computed using *static WCET analysis methods* that obtain WCETs through a static analysis of task source and/or object code [18].

WCET of programs is obviously influenced by the hardware in use, in particular the presence of caches. Caches are small and fast buffer memories containing recently referenced memory blocks. These blocks are likely to be accessed by the CPU in the near future thanks to temporal and spatial locality in reference streams. They are a very effective means of speeding up the memory accesses for the average case. However, the worst-case behavior of applications, which is of prime importance in hard real-time systems, is harder to predict in a safe and precise way when caches are used. The difficulty comes from 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 preemptions and imply a so-called *cache-related preemption delay* to reload the cache after a task has been preempted.

If caches are used without any restriction, real-time applications can fully benefit from the performance enhancement they provide. However, special attention is required to obtain deterministic guarantees for the system schedulability. At the task level, WCET analysis must be aware of the presence of caches. In order to have safe but accurate estimations of WCETs, WCET computation methods have to safely classify memory accesses into categories (for instance, for architectures without timing anomalies, *hit* when a memory access is guaranteed to be in the cache or *miss* otherwise). Cache-aware WCET computation methods have been designed during the last ten years [15, 1, 11, 12]. At the multitasking level, cache-related preemption delays have to be estimated as precisely as possible [9].

Other ways to face the predictability issue raised by caches are to find trade-offs between performance and predictability. Two main classes of methods may be used: *cache partitioning* and *cache locking*.

Cache partitioning techniques (e.g. [8, 19]) 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 thus eliminate the inter-task interferences, but one must still tackle intra-task interferences.

Cache locking techniques exploit hardware support allowing software to control the cache contents: *load* information into the cache and disable the cache replacement policy (*lock* or *freeze* the cache). This ability to lock cache contents is available in several commercial processors (ColdFire MCF5249, PowerPC 440, IDT79RC64575, ARM 940 and ARM 946E-S). The contents of the locked cache can be fixed for the whole execution of a task or changed at run-time. When cache reload points and contents of locked cache are computed off-line, cache accesses can be easily predicted statically.

Cache-aware WCET estimations methods provide tight estimations for direct-mapped caches, and for set associative caches with known and easy to predict cache replacement policy, such as the Least Recently Used (LRU) replacement policy. However, some replacement policies are not easily amenable to static WCET analysis, resulting is much less tight WCET estimates. For instance, Heckmann *et al* report in [6] that only 1/4 of the cache of the ColdFire MCF 5307 cache can be modeled, because of its pseudo-round-robin policy.

Moreover, in dynamically-scheduled processors, it is not safe to consider that a cache miss is the worst-case scenario anymore. Lundqvist and Stenström have exhibited in [13] *timing anomalies*, according to which a cache miss may in some cases result in a shorter execution time than a hit. Timing anomalies imply that to be safe, WCET estimation methods have to consider *all* outcomes for a memory access (hit/miss) when it cannot be guaranteed that the outcome is either a hit or a miss. Timing anomalies thus introduce some extra-complexity in the WCET estimation process.

When using static software control of caches, all memory accesses can be statically predicted whatever the replacement policy is. Furthermore, the timing anomalies described above do not occur anymore since cache contents is known off-line. The issue to be addressed then is to decide when the cache is reloaded as well as the associated contents, to obtain the "best performance". In the context of hard real-time systems, the most suited performance metric is *worst-case* performance since it serves at temporal validation and dimensioning of hardware resources. Thus we focus, at the task-level, at optimizing the task worst-case performance (WCET estimate).

Selecting cache reload points and associated cache contents in a blind manner (without any knowledge of application references) raises complexity issues even for very small programs (it would mean exploring all possible locations for reload points and all possible cache contents). Thus, information on memory access patterns of applications is helpful. Using profile data in the selection process may not always be appropriate since worst-case and average-case execution paths need not be the same. A more promising approach is to use worst-case execution path (WCEP). The difficulty in WCET-guided content selection is that when selecting information to be locked along the WCEP, a new path may become the WCEP (*instability* of WCEP). As a result, locally optimizing along the current worst-case path may not lead to the globally optimal solution.

Algorithms have been proposed to select the cache contents of locked cache in the case of a unique locked region spanning the whole task/system lifetime [14, 17, 4]. Obviously, the WCETs of programs using such locking schemes is comparable to the WCETs using unlocked caches only when the code/data size is of the same order of magnitude as the cache size. When the code/data size are much larger than the cache size, the WCET estimate increases dramatically [16]. These algorithms lack scalability with respect to code/data size.

In this paper, we propose two algorithms for software control of instruction caches, using cache locking. They introduce multiple reload points in the code of a program and select the values to be loaded into the cache such that the WCET estimate of the program is minimized. Multiple reload points are introduced for the sake of scalability of worst-case performance with the program code size. Two different algorithms are proposed:

- The first algorithm is a greedy algorithm with low complexity (linear with the number of basic blocks). It selects cache contents by exploiting execution frequencies of basic blocks along the WCEP. The stability issue is dealt with using a regular and customizable re-evaluation of the worst-case execution path.
- The second algorithm is a genetic algorithm exploring the search space (reload points + cache contents) in a blind manner.

Both algorithms place reload points at natural places for reuse in streams of references to instructions: loop and function entries. Both algorithms can be configured to select the maximum number of reload points: the larger the number, the better the worst-case performance, but the larger the memory occupied by reload points. This configurability allows to find a trade-off between consumed memory and worst-case performance.

Experimental results provided in the paper show that the worst-case WCET estimates of applications with locked instruction caches is close to the WCET estimates considering caches without software control (and in some cases even better), as far as applications exhibit temporal locality. The greedy algorithm yields acceptable WCET estimates with a very low run time. The genetic algorithm used as standalone is not very efficient when the initial population is selected at random, because of its long convergence time. However, it proves to be very efficient at improving the WCET estimate of applications using a locked cache those contents was selected by the greedy algorithm. Beyond worst-case performance considerations, locking techniques are an effective mean to limit the variability of execution times in architectures with caches. Locking techniques also

eliminate timing anomalies coming from caches, which result in extra-complexity in the WCET estimation process.

In summary, the main contribution of this work is in developing efficient techniques for allocating code portions in locked caches guided by the goal of reducing the program WCET estimate. Cache contents are changed at run time to ensure *performance* competitive with non-locked caches. Reload points and cache contents are computed off-line to ensure *predictability*.

The remainder of the paper is organized as follows. Section 2 presents the two algorithms for cache-contents selection. Their performance is evaluated in section 3. We compare our work with related work in Section 4. Finally, we conclude in Section 5 with a summary of the paper contributions and directions for future work.

2 Algorithms for software control of instruction caches

This section describes two algorithms for selecting the contents of locked instruction caches. A low-complexity greedy algorithm is presented in § 2.2. A more compute-intensive genetic algorithm is described in § 2.3.

2.1 Notations and assumptions

Without loss of generality, we consider a CPU with fixed-size instructions, with a W-way set-associative instruction cache and no prefetch mechanism. The cache is of size S_C and comprises a total of B blocks of S_B bytes each ($S_C = B * S_B$). 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 $\lfloor \frac{ad}{S_B} \rfloor mod\ S$. There are $ipcl$ instructions per cache block.

We consider that there exists a mechanism to *load and lock* blocks into the instruction cache, inhibiting cache replacement on those blocks until they are unlocked. In practice, hardware support for locking may provide means for locking either individual cache blocks or ways or the entire cache. Since we reload the whole cache at reload points, our algorithms can be used for these 3 classes of hardware support.

Our work considers an isolated task, represented by its control flow graph (CFG), statically extracted from its executable code. The task binary code is split into cache block-sized units named *instruction lines* hereafter. We term *pre-header* nodes the set of nodes directly preceding loop headers in the CFG. The body of every function is represented by an abstract loop with a number of iterations of one. Only reducible loops are currently supported.

The algorithms splits the task's code into *regions*. Each region has a unique associated *cache contents*. The cache contents is loaded and the cache is locked when entering the regions (at so-called *cache reload points*), until a subsequent cache reload point is encountered[1]. Reload points are placed at loop pre-headers, and cache contents comprizes only program lines of these loops. Furthermore, the locked regions altogether cover all the code of the task. As a consequence, it can be known statically if a given instruction line is a hit or a miss.

The user is offered the possibility to specify a maximum number of reload points, named *max_reload_points* hereafter, to find a balance between a low WCET estimate and and the space requirements to store the reload points.

The algorithms use the following constants. t_{hit} and t_{miss} are the hit/miss latencies; t_i and t_l represent the time required to reload the cache and lock it, expressed in number of misses; $t_i + t_l * pl$ cache misses are required to load and lock pl instruction lines into the instruction cache.

2.2 Greedy algorithm

To minimize the WCET estimate, the greedy algorithm relies on the knowledge of the number of executions of every basic block along the worst-case execution path (WCEP) at a given stage of the selection procedure. $f(bb)$ (resp. $f(pl)$) denote the total number of executions of basic block bb (resp. instruction line pl) along the WCEP.

The algorithm is made of two independent parts: selection of reload points (§ 2.2.1) and selection of cache contents (§ 2.2.2). The algorithm is illustrated on a small example in § 2.2.4.

2.2.1 Selection of reload points

Reload points are placed at loop pre-headers to exploit temporal locality. A cost function $CF(L)$, given in Equation 1, decides whether or not the cache should be reloaded at the pre-headers of a loop L. $CF(L)$ uses approximate values of the WCET of loop L respectively with a locked cache (*WCET_locked(L)*) and with an unlocked cache (*WCET_cache(L)*). *WCET_cache(L)* (see below) is an approximate version of the WCET of loop L, assuming only spatial locality is exploited. *WCET_locked(L)* is an approximate estimation of the loop WCET assuming the cache is loaded at the loop pre-headers with the most frequently executed instruction lines of the loop. It accounts for the execution frequencies of the instructions within the loop (hit or miss depending on whether the loop is larger or smaller than the cache) and the frequency of the reload point(s). $mfpl(L)$ contains the B most frequently executed instruction lines of loop L on the worst-case execution path in a system without any cache, with B the cache size. $pl(L)$ contains all instruction lines of L.

$$WCET\_cache(L) = \sum_{pl_i \in pl(L)} f(pl_i) * (t_{miss} + (ipcl - 1) * t_{hit})$$

[1] In multi-task applications, cache reloads occur at context switch times as well.

$$WCET\_locked(L) = \sum_{pl_i \in mfpl(L)} f(pl_i) * ipcl * t_{hit} + \sum_{pl_i \in pl(L) - mfpl(L)} f(pl_i) * ipcl * t_{miss} + \sum_{ph \in pre-head(L)} f(ph) * (t_i + t_l * |mfpl(L)|)$$

$$CF(L) = WCET\_cache(L) - WCET\_locked(L) \quad (1)$$

A positive value of $CF(L)$ means that the WCET of the loop whose content is locked at the loop pre-header(s) is expected to be lower than the WCET with an unlocked cache. The pre-headers of the *max_reload_points* loops with the maximum positive values of $CL(L)$ are selected as reload points.

2.2.2 Selection of cache contents

Selection of cache contents is based on frequency information along the WCEP. Since loading and locking a value into the instruction cache may change the WCEP, the WCEP is re-evaluated regularly, in a customizable manner. The algorithm for selection of cache contents is sketched below.

```
1  ToBePlaced = ListBasicBlocs;
2  evaluate_WCET(WCET,WCEP);
3  ListBB = SelectMostBeneficialBB(ToBePlaced,N);
4  while | ListBB | ≠ 0 do
5      for each BB in ListBB do
6          ListReloadPoints = getPoints(BB);
7          for each rp in ListReloadPoints do
8              for each instruction line pl in BB do Load(pl,rp);
9          end for
10     end for
11     evaluate_WCET(WCET,WCEP);
12     if WCET > WCET_previous_iteration return;
13     ListBB = SelectMostBeneficialBB(ToBePlaced,N);
14 end while
```

The algorithm fills progressively the cache contents associated with the regions identified in § 2.2.1, by considering successively all the program basic blocks. Initially (line 1), the set of basic blocks to be considered (*ToBePlaced*) includes all basic blocks of the program. All reload points initially have an empty cache contents and will be filled-in progressively as explained below.

The algorithm proceeds iteratively. At a given iteration, the group formed by the N most *beneficial* basic blocks are considered for locking. The notion of benefit of a basic block for locking (function *SelectMostBeneficialBB*) is based on a cost function $CF(bb, L)$ given in equation 2. In the equation, L is a reload point where bb may be loaded and $|bb|$ denotes the size of basic block bb in bytes. The cost function $CF(bb, L)$ is positive when the locking of basic block bb is expected to have a lower WCET estimate than in a system without a cache (first line of the equation) and do not degrade too much the WCET estimate as compared to a system with a cache (second line in the equation).

$$WCET\_cache(bb, L) = \sum_{pl_i \in PL(bb)} f(pl_i) * (t_{miss} + (ipcl - 1) * t_{hit})$$

$$WCET\_nocache(bb, L) = \sum_{pl_i \in PL(bb)} f(pl_i) * ipcl * t_{miss}$$

$$WCET\_locked(bb, L) = \sum_{pl_i \in PL(bb)} f(pl_i) * ipcl * t_{hit} + \sum_{ph \in pre-head(L)} f(ph) * (t_i * \frac{B}{|bb|} + t_l * |bb|)$$

$$CF(bb, L) = (WCET\_nocache(bb, L) - WCET\_locked(bb, L)) + (WCET\_cache(bb, L) - WCET\_locked(bb, L)) \quad (2)$$

The inner loop of the algorithm (lines 6 to 9) evaluates the locking of all instruction lines of the basic block *bb* under consideration. First we get the list of reload points at which *bb* may be loaded (line 6); function *getPoints*, not detailed here for space considerations, returns the list of reload points corresponding to loops/functions in which bb is included. For every instruction line pl of bb and every reload point rp the line is loaded and locked (line 8).The algorithm iterates until locking new instruction lines do not result in improvements of WCETs anymore (line 11 and 12).

The WCEP and the cost function are re-evaluated regularly, after having considered the placement of group of N basic blocks (line 13). The lower the value of N the better the estimation of the WCEP along the whole algorithm and the better the quality of the cache contents (but the longer the execution time of content selection). Parameter N may then serve to find a balance between the quality of the solution and the computation time required to obtain it.

2.2.3 Optimality and complexity considerations

Although not proven formally, the similarity of the problem of reload point and cache contents selection with the problem of register allocation in compilers make us suspect the problem to be NP-hard. As a consequence, the method we propose is based on heuristics. Moreover, it can be shown on a simple example that the selected cache contents, although resulting in tight WCET estimates in practice, is not optimal.

The most time consuming operation of the greedy algorithm is the evaluation of the WCET and WCEP (this operation is complex because it requires an in-depth analysis of the program structure and as well as its interaction with the underlying architecture). As a consequence, it is natural to express the complexity of our algorithms in terms of number of WCET estimations. Selection of reload points requires exactly one WCET estimation. Selection of cache

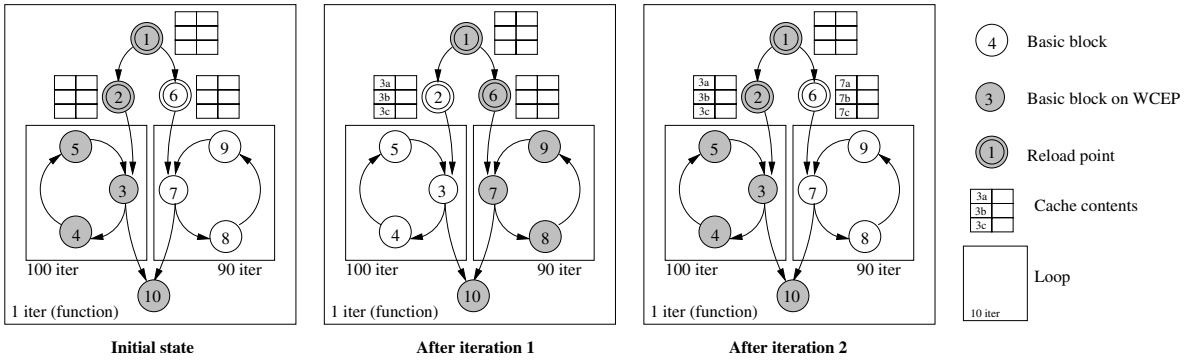

Figure 1. The greedy algorithm on an example

contents require $\frac{NbBB}{N}$ WCEP estimations, where $NbBB$ is the number of basic blocks and N is number of basic blocks considered before re-evaluating the WCEP. Thus at worst, $NbBB$ WCEP estimations are required for cache content selection, corresponding to the most frequent re-evaluation of the WCEP. Measured execution times of the selection procedure will be given in section 3.

2.2.4 Example

Figure 1 depicts two iterations of the greedy algorithm on a toy example, with $N = 1$ (the WCEP is evaluated every time a basic block is considered for locking). The initial view depicts the locations where reload points are placed by the algorithm presented in 2.2.1. The initial WCEP is assumed to traverse the loop at the left. Assuming the basic block with the largest value of CF(bb,l) is BB number 3, its instruction lines (3a, 3b and 3c) are locked in the region starting at BB 2 (loop pre-header). Due to the effect of locking 3a, 3b and 3c, the WCEP then changes. Assuming the BB with the highest value of CF(bb,l) is then BB 7, its instruction lines are locked, etc.

2.2.5 Implementation considerations

Code must be executed at every reload point to load the cache with a new cache contents and lock it. Two implementation alternatives may be considered for invoking the cache reload code. One alternative is to use the processor debugging capabilities, if any, to trigger the execution of the cache reload routine. It exploits the processor ability to raise an exception (breakpoint exception) when a specified instruction, whose address is loaded in the processor debug register(s) is encountered. No compiler support is required, and the task's memory map is not changed, but the approach is not portable. Another alternative is to use compiler support. One way is to insert a call to a cache reload routine at every reload point. Inserting function calls changes the task's memory map, but this is not a problem since cache reload points and cache contents are selected independently. Another way is to modify the task binary code at the reload point, in such a way that the original code is replaced by a call to the cache reload routine; the cache reload routine reloads and locks the cache and then executes the original code.

2.3 Genetic algorithm

The second algorithm we have devised for cache control does not use any knowledge about the software memory access pattern. It uses a genetic algorithm for both selection of reload points and selection of cache contents. Genetic algorithms are inspired by Darwin's theory of evolution, and are particularly suited to the resolution of optimization problems with a very large search space, which is the case in the problem under consideration. Genetic algorithms operate on a population of potential solutions applying the principle of survival of the fittest to produce better and better approximations of a solution. At each generation, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics.

The use of genetic algorithm in any search problem requires the definition of a set of elements and operators: representation of the solutions (*codification*), a *fitness function* to evaluate the different solutions, a *selection scheme* to sort candidate individuals for breeding, *cross-over* and *mutation* operators to transform the selected individuals.

Codification. Each individual, representing a possible solution, is an array of chromosomes each of them being a pair $(rp, contents)$. rp is an identification of the reload point and $contents$ is the associated cache contents.

Fitness. The fitness function is simply the WCET.

Selection. The probability to select one individual for breeding is a linear function of its fitness value (here, WCET).

Crossover and mutation. One point crossover is applied: an index into the parents chromosomes (array of pairs *(reload point, contents)*) is randomly selected. All

Name	Description	Code size (bytes)	WCRN
des	des and triple-des encryption/decryption algorithm	11068	41407400
adcpm	Adaptive differential pulse code modulation	8504	124240
minver	Matrix inversion for 3x3 floating point matrices	4520	113540
fft	Fast Fourier Transform	3524	822880
compress	Compression of a 128 x 128 pixel image using discrete cosine transform	3056	138129850
nsichneu	Simulation of an extended Petri Net. Automatically generated code containing large amounts of if-statements	45720	1940910
flight	Control code (flight control) mixing floating point computation, switches and conditional statements, automatically generated code from the SCADE suite.	9944	180350

Table 1. Task characteristics

data beyond that point in the chromosomes are swapped between the two parent organisms, defining the children chromosomes. Three types of mutations have been introduced and are applied to individuals with a user-selected probability: M_{rem} (removal of one reload point selected randomly, M_{add} (addition of one reload point selected randomly), M_{chg} (change of the contents of a reload point selected randomly). A fixed number of instruction lines are replaced by other (not yet locked) instruction lines selected randomly.

Initial population. The initial population is made of a fixed number of individuals. Every individual has a random number of (unique) reload points. For each reload point the associated cache contents is selected randomly (without duplicated program lines). The higher a loop in the loop nesting hierarchy, the higher the probability for selecting the loop as a reload point. The lower a loop in the loop nesting hierarchy, the higher the probability for the selection of its basic blocks in a locked cache contents. We also tested an initial population the individuals are results of greedy algorithm presented before for different values of *max_reload_points*.

An interest of genetic algorithms is that the produced results (here, cache contents) can be used at any time. Their limit is their computation time, as we will be shown in Section 3.

3 Experimental results

3.1 Experimental setup

Our interest here is to evaluate the predictability of programs with respect to the memory hierarchy. As a consequence, in order to not interfere with other micro-architecture elements, our performance metrics are limited to the number and/or ratios of hits/misses in the instruction cache. All the numbers given in this section are **worst-case** numbers, obtained using a static WCET analysis tool. The term *miss ratio* will then denote the miss ratio obtained by a cache-aware WCET estimation tool and not a measured miss ratio.

Our experiments were conducted on MIPS R2000/R3000 binary code, but we are actually independent of any specific MIPS-compatible processor since our focus is on instruction caches only. We consider an instruction cache with blocks of size $S_B = 16$ bytes ($ipcl$=4 instructions), and an associativity degree of 4. The cache replacement policy is either Least Recently Used (LRU), tightly analyzable by cache-aware WCET analysis tools, or Pseudo-Round Robin (PRR), shown in [6] to be analyzable for 1/4 of the cache only. By default, the cache size is $1KB$, $t_{hit} = 1$, $t_{miss} = 10$, $t_i = 0$, $t_l = 1$.

The WCETs of tasks are computed by the Heptane[2] static WCET analysis tool [5]. One may configure Heptane to estimate WCETs using either: a tree-based method, through a bottom-up traversal of the syntactic tree of the analyzed C programs; an IPET (Implicit Path Enumeration Technique) method, generating a set of linear constraints from the program control-flow graph. Here, the IPET WCET estimation method is used.

Heptane includes hardware modeling capabilities to estimate WCETs for programs running on architectures with instruction caches. The technique used in Heptane to estimate the worst-case behavior of applications with respect to the instruction cache (see [5] for details) is based on F. Mueller's *static cache simulation* [15]. The cache analysis technique computes *abstract cache states* (representation of all the possible cache contents considering all the possible execution paths in the program) using data-flow analysis on the program control flow graph. Abstract cache states are then used to safely classify the instructions according to their worst case behavior regarding the instruction cache (i.e. *hit*, *miss*, *first-hit*, *first-miss* the latter two categories being used for instructions in loop bodies [5]). Heptane was modified by replacing its cache analysis module by a module classifying instructions according to their presence in a locked region.

The experiments were conducted on seven benchmark tasks, whose features are summarized in Table 1. The last column (WCRN) counts the number of references to instructions along the WCEP. This information gives an in-

[2]Heptane is an open-source static WCET analysis tool available at *http://www.irisa.fr/aces/software/software.html*.

dication of the amount of loops in the benchmark.

All benchmarks but flight and compress are benchmarks maintained by the Mälardalen WCET research group[3]. Compress is from the UTDSP Benchmark suite[4].

The default parameters for the greedy algorithm is $N = 10$ (re-evaluation of WCEP after placing 10%) of the basic blocks. By default, unless explicitly stated, we do not restrict the maximum number of reload points (*max_reload_points*=20).

3.2 Evaluation of the greedy algorithm

Table 2 and Figure 2 compare hit and miss ratio for locked and unlocked caches, as well as the percentage of memory accesses required for cache reload ($\frac{n_{miss\_reload}}{n_{hit}+n_{miss}}$).

Task	Hit ratio	Miss ratio	Reload ratio	Nb points
des LRU	92.1%	7.9%	0.0%	–
des PRR	83.0%	17.0%	0.0%	–
des locked	86.8%	13.2%	3.7%	14
adcpm LRU	92.2%	7.8%	0.0%	–
adcmp PRR	91.8%	8.2%	0.0%	–
adcpm locked	71.3%	28.7%	2.5%	15
minver LRU	92.4%	7.6%	0.0%	–
minver PRR	82.1%	17.9%	0.0%	–
minver locked	70.0%	30.0%	15.9%	16
fft LRU	91.7%	8.3%	0.0%	–
fft PRR	75.3%	24.7%	0.0%	–
fft locked	90.8%	9.2%	7.6%	8
compress LRU	99.7%	0.3%	0.0%	–
compress PRR	95.5%	4.5%	0.0%	–
compress locked	96.8%	3.2%	0.8%	13
nsichneu LRU	73.2%	26.8%	0.0%	–
nsichneu PRR	73.2%	26.8%	0.0%	–
nsichneu locked	2.2%	97.8%	0.0%	1
flight LRU	73.4%	26.6%	0.0%	–
flight PRR	73.3%	26.7%	0.0%	–
flight locked	16.0%	84.0%	0.4%	1

Table 2. Hit/miss/reload ratios for locked & unlocked caches (ratios=$\frac{n_{miss/hit/reload}}{n_{miss}+n_{hit}}$, %)

For applications with both temporal and spatial locality (all but the last 2 ones), in all cases the hit ratio for locked cache is close to the hit ratio with an unlocked cache. When a predictable replacement policy is considered (LRU), the miss ratio with an unlocked cache is always lower than with a locked cache. When a hard to predict replacement policy is used (PRR) for most applications the miss ratio with an locked cache is lower than with an unlocked cache. This shows the interest of locking schemes when using caches with hard-to-predict replacement policies.

For applications with mostly spatial locality and poor temporal locality (no loops or loops whose body is much larger than cache size), the miss ratio with a locked cache is high. It comes from the locations where reload points are placed, which exploit temporal locality only[5]. The idea behind restricting reload points to loop entries is that the cost for cache reload (which is necessarily high since cache reload is under software control) is masked by the multiple iterations of the loop. Placing reload points inside loop bodies would not necessarily exploit spatial locality because of the cost of cache reloading, which would be prohibitive if the cache was reloaded at every loop iteration. How to better exploit spatial locality is left as future work. Perhaps minimal hardware support (an instruction line-size buffer intended to take benefit of spatial locality) would be a good trade-off between predictability and performance.

The percentage of extra memory accesses required for cache reload ($\frac{n_{miss\_reload}}{n_{hit}+n_{miss}}$) is depicted as black bars in Figure 2. The percentage of extra memory accesses is in general low. This indicates that the cost function used for selecting of reload points succeeds in detecting loops for which the cache reload overhead is minimal. The highest percentage of memory accesses due to cache reloads is for applications with nested loops (*minver* and *fft*) for which the algorithm places reload points inside the outer loop to exploit temporal locality within the inner loops.

Notice that the number of reload points actually inserted is always lower than *max_reload_points*, because reload points are inserted only if they result in a WCET improvement.

Stability issues.

The task's WCET estimate depends on how often the WCEP is re-evaluated during the contents selection process (parameter N in the algorithm). The impact on parameter N is studied in tables 3 and 4.

Task	100	50	30	20	10	5	2	0
des	18.4	17.4	16.3	16.6	16.3	16.3	16.7	19.9
adcpm	45.6	45.6	45.6	45.6	42.9	41.0	41.1	41.1
minver	37.3	36.4	36.8	37.4	39.5	44.3	34.0	34.0
fft	15.7	15.6	15.7	15.8	15.6	12.7	15.3	15.3
compress	4.2	4.2	4.2	4.2	4.0	3.9	3.3	3.3
nsichneu	97.8	97.8	97.8	–	–	–	–	–
flight	84.1	84.1	84.1	–	–	–	–	–

Table 3. Miss ratio ($\frac{n_{miss}+n_{miss\_reload}}{n_{miss}+n_{hit}+n_{miss\_reload}}$, %) in function of N

Table 3 gives the miss ratio (reload accesses included) for different values of N. To be application independent, N is expressed as the percentage of basic blocks considered between two re-evaluations of the WCEP (a value of 0 means that the WCEP is re-evaluated every time a basic block is considered for locking). The numbers in table 3 show that re-evaluating the WCEP in the course of contents selection is required. However, a too frequent re-evaluation

[3] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[4] http://www.eecg.toronto.edu/

[5] It's a coincidence that the applications with the worse WCET estimate are those with the larger code size. Smaller applications with the same kind of memory access patterns had the same bad WCETs.

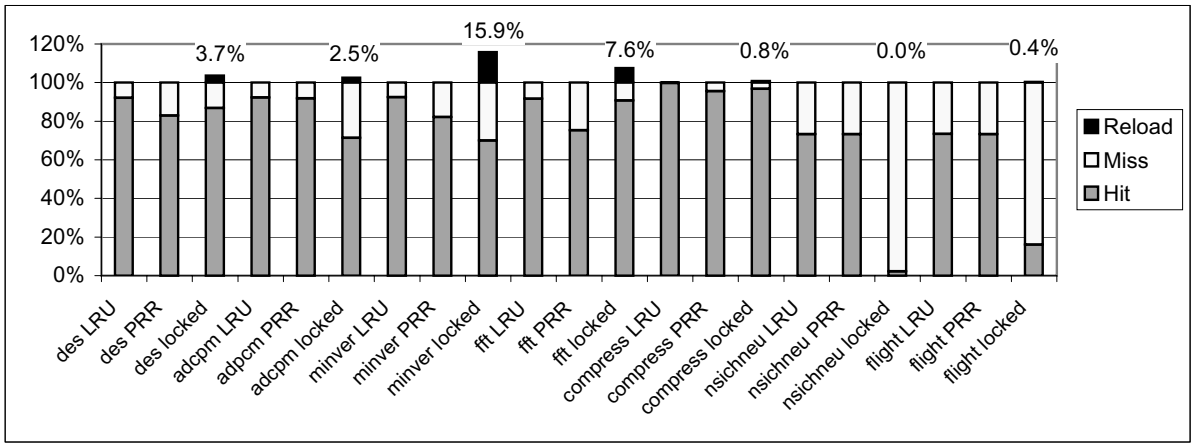

Figure 2. Hit/miss/reload ratios for locked & unlocked caches (ratios=$\frac{n_{miss/hit/reload}}{n_{miss}+n_{hit}}$, %)

of the WCEP results in a negligible decrease of the miss ratio, when considering the extra-time required to re-evaluate the WCEP (see table 4, showing that time for content selection is approximately linear with N).

Appli	100	50	30	20	10	5	2	0
des	63	98	141	140	224	308	556	833
adcpm	63	101	190	237	453	229	613	1635
minver	19	30	40	37	37	36	84	145
fft	28	45	79	78	129	195	105	186
compress	19	26	47	58	57	57	102	189
nsichneu	455	830	1586	–	–	–	–	–
flight	1365	1964	3225	–	–	–	–	–

Table 4. Running time of contents selection (seconds) in function of N

Space vs worst-case performance trade-off.

As cache contents is under software control, the instruction lines to be loaded at run time have to be stored in the application binary. Thus it may be interesting to control the amount of memory required to store cache contents. Tables 5 and 6 address this issue. Tables 5 and 6 give respectively the required memory and miss ratio when the maximum number of reload points (*max_reload_points*) varies between 20 and 1 (one unique locked region per task).

Appli	20	15	10	7	5	3	2	1
des	11424	11424	4784	2672	2752	1136	1184	1024
adcpm	3728	3728	2400	1376	1296	1120	1104	1024
minver	8032	5888	3952	2736	2256	1664	1328	1024
fft	4944	4944	4944	2640	1616	1440	1136	1024
compress	1344	1344	1344	1312	1312	1024	1024	1024

Table 5. Space for storing cache contents (Bytes) in function of *max_reload_points*

Appli	20	15	10	7	5	3	2	1
des	16.3	16.3	16.6	17.5	19.9	32.1	45.7	59.5
adpcm	30.4	30.4	36.1	41.0	42.3	46.9	47.8	49.1
minver	39.5	38.4	33.8	35.1	37.1	44.9	46.8	52.0
fft	15.6	15.6	15.6	14.2	13.1	13.7	34.4	34.0
compress	4.0	4.0	4.0	4.0	4.3	3.8	3.8	3.8

Table 6. Miss ratio ($\frac{n_{miss}+n_{miss\_reload}}{n_{miss}+n_{hit}+n_{miss\_reload}}$, %) in function of *max_reload_points*

The numbers given in Tables 5 and 6 show that reducing the number of regions increases the miss ratio but saves memory. The function linking memory savings and miss ratio decrease is not linear. On some applications like *des*, one can largely reduce memory consumption with only a small increase of miss ratio. For embedded systems where memory demand is as important as real-time performance, the results given in Tables 5 and 6 can be exploited to find the best trade-off between space and worst-case performance.

Table 6 indirectly shows that the locking scheme proposed in the paper takes benefit of caches even if the code size is larger than the cache size. For instance, there are 83.7% of hits for the *des* application whose code size is 11 times the cache size. In contrast, locking techniques that lock the cache using one unique region for the whole application exhibit a much lower hit ratio (40.5% for *des* if a unique locked region is used).

3.3 Evaluation of the genetic algorithm

The parameters of the genetic algorithm used for the experimental evaluation of the genetic algorithm are: a population of 32 individuals, 40 generations, and probabilities of mutations of 10% (mutation M_{rem}), 10%(mutation M_{add}), 80% (mutation M_{chg}). We have tested the genetic algorithm with two initial populations: (i) a population of individuals

selected randomly; (ii) a population of individuals made of solutions given by the greedy algorithms, for different values of parameter *max_reload_points*.

We did not further investigate the first alternative: roughly 2 hours of computation were required to find a solution as good as the one given by the greedy algorithm for the *des* application, against one minute for the greedy algorithm. The results of the second alternative are given in Table 7. The columns give the miss ratio for unlocked caches (LRU and PRR in col. 1 and 2), locked caches (greedy and genetic in col. 3 and 4). Column 5 gives the execution time of the genetic algorithm.

Appli	Cache LRU	Cache PRR	Greedy	Genetic	Time (mn)
des	7.9	17.6	16.3	14.1	323
adpcm	7.8	8.2	30.4	30.2	300
minver	7.8	17.9	39.5	24.1	181
fft	8.3	24.7	15.6	11.8	217

Table 7. Performance of genetic algorithm (miss ratio $\frac{n_{miss}+n_{miss\_reload}}{n_{miss}+n_{hit}+n_{miss\_reload}}$, %)

In all applications the miss ratio obtained using the greedy algorithm has been significantly reduced by using the genetic algorithm. The drawback is the time required to execute the genetic algorithm. Thus, genetic algorithms should be reserved to fine-tune performance in the last steps of software production.

4 Related work

Several cache-aware WCET computation techniques have been designed in the last decade, mainly for instruction caches. For architectures without timing anomalies, their objective is to determine for every memory access if it *will certainly* cause a cache hit or *may* cause a miss. Cache-aware WCET computation methods can be based on data-flow analysis [15], abstract interpretation [6], integer linear programming techniques [11], or symbolic execution [12]. In comparison, much less work has tackled data caches because of the presence of dynamic references (i.e. arrays and pointers).

As mentioned in the introduction, cache-aware WCET analysis techniques reach their limit for some cache replacement policies (e.g. random, pseudo round-robin, pseudo LRU), which cannot be tightly predicted, and for dynamically scheduled processors, for which timing anomalies arise. In such situations, a statically-decided software control of the cache, provided by cache locking techniques, is of interest.

Work on locking techniques has been done for instruction caches [17, 4] and data caches [22]. [17, 4] study locking techniques of instruction caches in the case the cache contents for a given task does not change at run-time. While such techniques provide good worst-case performance for small tasks, their performance decreases dramatically when the task working set exceeds the cache size. The work presented in this paper is a follow-up of this line of research and provides a much better scalability with task code size.

Vera *et al* introduce in [22] a set of methods for making data caches predictable. Instead of using locked caches all over the execution of tasks like in our proposal, they combine the use of dynamic caches (together with cache analysis) with a localized use of cache locking for the parts of code where cache analysis fails (e.g. data-dependent conditionals). The selection of cache contents for locked regions is achieved using an extension of Cache Miss Equations. Their approach for cache contents selection is different to our approach in the sense their objective is predictability only, without any further attempt to reduce the WCET estimate. Moreover, since they make a localized use of cache locking, knowledge of the cache replacement policy is required in their approach.

A last approach to address the predictability issue raised by caches is to design more predictable caches. For instance, [7] proposes to extend the cache hardware and to introduce new instructions to control cache replacement (kill or keep cache blocks). In contrast to this class of approaches, our work uses standard hardware support to load and lock the cache rather than defining new ways to control the cache contents and cache replacement.

The predictability issue raised by caches may lead some designers of hard real-time systems to avoid the use of caches, or to employ on-chip static RAM (scratchpad memories) instead [2]. The amount of on-chip scratchpad memory available is often small compared to the total amount of cache memory available in (potentially multi-level) cache hierarchies. However, accesses to scratchpads are predictable since transfers to and from scratchpad memories are under software control, and the power consumption of the system is definitely lower than that of a system using a cache [3]. Scratchpad are thus an interesting alternative to the use of caches.

The issue of selecting which information is loaded into scratchpad is very close to deciding which information has to be locked into a cache. The issue has been studied from different perspectives: reduction of energy consumption [23, 20], average execution time [10], worst-case execution time. To the best of our knowledge, only [21] addresses the problem of allocating information in scratchpad memory with the objective of reducing the WCET estimate. Their method allocates data for the whole lifetime of a program, which should result in scalability problems for program whose data size is larger than scratchpad memory size.

5 Concluding remarks

We have proposed algorithms to use instruction caches in a predictable manner in real-time systems. The proposed algorithms statically divide the code of tasks into regions, for which a cache contents is statically selected. At run-time, for every transition between regions, the cache con-

tents computed off-line is loaded into the cache and the cache replacement policy is disabled (the cache is *locked*). Performance results provided in the paper show that the worst-case performance of applications with locked instruction caches is close to the worst-case performance of caches without software control for programs with temporal locality, but fails at fully exploiting spatial locality.

Dealing with hierarchies of instruction caches is a straightforward extension of our work that we intend to explore in the near future. Extending the work to data caches is trickier, in particular detecting locality regions for scalars. A similar problem we are interested in is the loading of code/data into scratchpad memories to improve worst-case performance. More generally, compiling for predictability and worst-case performance is an interesting research area we wish to explore.

Acknowledgments

Many thanks to André Seznec, Jan Staschulat and Eric Petit for fruitful feedback on earlier drafts of this paper. The genetic algorithm was developed by Gael Legargeant during a summer internship.

References

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

[2] O. Avissar, R. Barua, and D. Stewart. Heterogeneous memory management for embedded systems. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Atlanta, USA, Nov. 2001.

[3] L. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory : a design alternative for cache on-chip memory in embedded systems. In *Proceedings of Tenth International Workshop on Hardware/Software Codesign (CODES 2002)*, May 2002.

[4] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix. Cache contents selection for statically-locked caches: An algorithm comparison. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 49–56, Palma de Mallorca, Spain, July 2005.

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

[6] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7), July 2003.

[7] P. Jain, S. Devadas, D. W. Engels, and L. Rudolph. Software-assisted cache replacement mechanisms for embedded systems. In *ICCAD*, pages 119–126, 2001.

[8] 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, Dec. 1989.

[9] 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.

[10] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[11] 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, Dec. 1996.

[12] 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, Nov. 1999.

[13] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

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

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

[16] I. Puaut, A. Arnaud, and D. Decotigny. Performance analysis of static cache locking in hard real-time multitasking systems. Technical Report 1568, IRISA, Oct. 2003.

[17] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, pages 114–123, Austin, Texas, Dec. 2002.

[18] 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.

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

[20] 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 *Proceedings of the 15th International Symposium on System Synthesis (ISSS 2002)*, pages 213–218, Kyoto, Japan, 2002.

[21] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS05)*, Dec. 2005.

[22] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics 2003)*, 2003.

[23] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of Design Automation and Test in Europe (DATE)*, Paris, France, Feb. 2004.