Proceedings of the 17th Euromicro
Conference on Real-Time Systems.
Pages 49-56. Palma de Mallorca, Spain,
July 2005

Cache contents selection for statically-locked instruction caches: an algorithm comparison

Antonio Martí Campoy*, Isabelle Puaut**, Angel Perles Ivars*, Jose Vicente Busquets Mataix*
* Computer Engineering Department, Technical University of Valencia, Spain
** Université de Rennes / IRISA, 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 or may be too pessimistic. An alternative approach allowing to use caches in real-time systems is to lock their contents (i.e. disable cache replacement) such that memory access times and cache-related preemption times are predictable. In this paper, we compare the performance of two algorithms for static locking of instruction caches: one using a genetic algorithm *for cache contents selection [3] and a pragmatical algorithm, called herafter* reference-based *algorithm [14], which uses the string of memory references issued by a task on its worst-case execution path as an input of the cache contents selection algorithm. Experimental results show that (i) both algorithms behave identically with respect to the system worst-case utilization; (ii) the genetic algorithm behaves slightly better than the reference-based algorithm with respect to the average slack of tasks; (iii) the execution time of the cache-contents selection procedure is much better when using the reference-based algorithm than with the genetic algorithm.*

1 Introduction

1.1 Caches and hard real-time systems

Cache memories have been designed as a solution for the ever-growing discrepancy of speed between processors and relatively slow memory components. These are typically small content associative memories with small access time, which are inserted between the CPU and the main memory and which act as ante-memories. Indeed no change is required in the memory addressing scheme since caches act transparently. They work in such a way that they exploit the spatial and temporal locality of memory reference streams. Therefore at any time cache memories contain memory blocks that are likely to be accessed in the near future. A key property of cache memories is that they improve the average performance of a computer system.

A real-time system is a computer system for which the good functioning is not only defined by the correctness of results, but also by the dates at which these results are to be produced. More particularly, a hard real-time system requires the exact knowledge of these dates. In order to satisfy this requirement, such a system must use an adequate scheduling policy such as Fixed Priority Preemptive (FPP), and has to be validated. The validation process consists of the computation of the worst-case execution time (WCET) of each task, and then in performing a schedulability analysis on the whole task set. Extensive studies have been done on both latter subjects.

Now cache memories are a source of unpredictability issues. Two phenomena consolidate this fact :

- Intra-task interference, which occur when a task overrides its own blocks in the cache, due to conflicts.
- Inter-task interference, which arise in multitasking systems, due to preemptions. These interference imply a so-called cache-related preemption delay to reload the cache after a task was preempted.

As a consequence, the designer of a hard real-time system may choose not to use cache memories at all, at the cost of over-sizing the system, or may choose to use scratch-pad memories [1], which are basically on-chip static memories. Nevertheless there is a growing demand in the industry of hard real-time systems with better performance and cheaper hardware. This fact drives to consider processors with cache

hierarchies. As regards the validation process, an important issue here is to cope with the effects of cache memories.

There are at the present time two categories of approaches to deal with caches in real-time systems. In the first one, cache analysis, caches are used without any restriction. Static analysis techniques (cache-aware WCET analysis [11, 9] and schedulability analysis [2, 6]) predict their worst-case impact on the system schedulability.

The second category of approaches consists in using caches in a restricted or customized manner so as to adapt them to the needs of real-time systems and schedulability analysis. Cache partitioning techniques [5] assign reserved portions of the cache to certain tasks in order to guarantee that their most recently used code or data will remain in the cache while the processor executes other tasks. The dynamic behavior of the cache is kept within partitions. These techniques eliminate inter-task interference, but need extra support to tackle intra-task interference (e.g. static cache analysis) and reduce the amount of cache memory available for each task.

Within the same category, another way to deal with caches in real-time systems is to use *cache locking* techniques [3, 14, 19], which load the cache contents with some values and lock it to ensure that the contents will remain unchanged. This ability to lock, entirely or partially, the cache contents is available on several commercial processors (among others : Motorola ColdFire MCF5249, Motorola PowerPC 603e, IDT RC64575, ARM 940). The cache contents can be loaded and locked at system start for the whole system lifetime (static cache locking), or changed at run-time, like for instance when a task is preempted by another one (dynamic cache locking). The key property of cache locking is that the time required to access the memory is *predictable*.

1.2 Paper contributions and organization

Regarding static locking of instruction caches, two classes solutions have been proposed: one using a *genetic algorithm* for cache contents selection [3] and a pragmatical algorithm, called hereafter *reference-based* algorithm, which uses the string of memory references issued by a task as an input of the cache contents selection algorithm [14]. These two classes of solution can be tailored to select the cache contents at the system level (global locking) or at the task-level (local locking) with then the necessity of reloading the cache contents at context switch points. In this paper, we concentrate on global locking.

This paper is devoted to a comparison of these two classes of solutions (genetic and reference-based) for instruction cache locking. Experimental results show that (i) both algorithms behave identically with respect to the system worst-case utilization; (ii) the genetic algorithm behave slightly better than the reference-based algorithm with respect to the average slack of tasks; (iii) the execution time of the cache-contents selection procedure is much better when using the reference-based algorithm.

The rest of this paper is organized as follows. Section 2 first briefly presents the two compared algorithms for cache contents selection. Section 3 presents the experimental setup. Section 4 then achieves a statistical analysis of the performance of systems using the two cache-contents selection algorithms. The comparison metrics are the worst-case processor utilization, the task slacks and the execution times of the cache contents selection algorithms. We present some concluding remarks in Section 5.

2 Cache contents selection algorithms

After a presentation of considered assumptions and notations (§ 2.1) this section briefly presents the compared cache contents selection algorithms (§ 2.2 and 2.3).

2.1 Assumptions and notations

In the description of the algorithm, we consider a CPU equipped with a W-way set-associative instruction cache. The cache comprises a total of B blocks of S_B bytes each. 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 $\lfloor \frac{ad}{S_B} \rfloor mod\ S$. Our cache model encompasses *direct-mapped* caches ($W = 1, S = B$), *set-associative* caches ($1 < W < B$) and *fully-associative caches* ($W = B, S = 1$).

The two algorithms presented hereafter consider a set of N periodic tasks $\tau_{i,\,1\leq i\leq N}$ with periods T_i and deadlines D_i. The worst-case execution time of task τ_i is noted C_i. The code of a task τ_i is split into *program lines* $L_{i,j}$ of size S_B; a *program line* is a sequence of instructions mapped into a cache block. We consider a fixed-priority scheduling for tasks, and tasks are numbered according to their respective priorities, task τ_0 being the task with the highest priority.

2.2 Reference-based algorithm [14]

We assume that for every task τ_i the sequence σ_i of program lines issued by τ_i is known. We call $n_{load_{i,j}}$ the number of references to program line $L_{i,j}$ issued by τ_i in the sequence σ_i. The algorithm uses this information to select the cache contents, and aims at minimizing the worst-case CPU utilization $\sum_{i=1}^{N} \frac{C_i}{T_i}$.

The very simple idea behind the algorithm is to lock the most frequently referenced program lines of every task. As multiple program lines from different tasks may compete for the same cache block, priority is given to the tasks with the greater importance I_i. In the context of this paper, focusing on periodic tasks, I_i is defined as the inverse proportional to its period T_i, which corresponds to the static priority assignment used by the Rate-Monotonic (RM) scheduling policy [8], optimal among static priority assignments for the considered task model. However, different (static) definitions of the importance of tasks can be used as well, such as the deadline monotonic static priority assignment [7] or user-defined task importances.

> The algorithm works as follows. Let L_s be the set of program lines (for all possible tasks) that can be mapped into the W cache blocks blocks of set s. The algorithm locks into the W blocks of every set s the W program lines $L_{i,j}$ of L_s having the highest $n_{load_{i,j}} * I_i$ factor.

Note that the choice of the sequences σ_i has an impact on the contents of the locked cache, and therefore on the system performance. Indeed, since task τ_i can follow different paths according to its input data, σ_i depends on the considered execution path. In this paper, the algorithm is applied on the sequences of memory references issued along the worst-case execution path of every task assuming no cache is used.

2.3 Genetic algorithm [3]

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. They 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. This process leads to the evolution of populations of individuals that are better suited to their environment than their ancestors, just as in natural adaptation.

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 a bitmap of size nl, with nl the total number of program lines for all tasks. A bit set to 1 means that the corresponding program line is locked into the cache.

Fitness. The fitness function is the weighted average of all tasks response times (see equation 1, where R_i denotes the response time of task τ_i. This fitness function aims at improving the average response-time of tasks, precluding that the genetic algorithm assigns *all* availaible cache blocks to the higher priority tasks. Other fitness function has been evaluated in [17] and have shown slight improvements on some performance metrics.

$$Fitness = \frac{R_0 + R_1 + 2R_2 + 4R_3 + ... + 2^{N-2}R_{N-1}}{2^{N-1}} \quad (1)$$

Selection. Rank-based selection, in which the probability to select one individual is a linear function of its rank [10], is used. Individuals are sorted according to both their fitness value and their validity (an individual is said to be *valid* if the number of bits set in its representation is lower or equal to the number of cache blocks B).

Crossover and mutation. One point crossover is applied: an index into the parents chromosomes is randomly selected. All data beyond that point in the chromosomes is swapped between the two parent organisms, defining the children chromosomes. Three types of mutations have been introduced:

- M1. random reduction of the number of locked program lines,
- M2. random increase of the number of locked program lines,
- M3. random modification of the identity of one locked program line, the total number of locked program lines being left unchanged.

Rule M1. (resp M2.) applies to invalid individuals whose number of locked program lines is greater than (resp. lower than) B, while rule M3 applies to valid individuals.

Initial population and algorithm parameters. The initial population is made of valid individuals only. The bitmap of every individual in the initial population has B consecutive bits set, the index of this series of 1 being randomly selected. The other parameters of the algorithm are given in table 1.

These parameters spring from a set of experiments where the behaviour of the genetic algorithm was studied.

Population size	200
Number of generations	5000
Probability of crossover	0.6
Probability of mutation (rules R1..R3)	0.01
Probability of selection of the individual with the highest rank	0.1

Table 1. Parameters of the genetic algorithm

An interest of genetic algorithms is that the produced results (here, cache contents) can be used at any time, that is, it is not necessary to wait the algorithm end to get partial results.

3 Experimental setup

3.1 Hardware configuration

We consider a processor with an instruction cache and a 16B (4 instructions) instruction prefetch buffer. The cache configurations used in the performance comparison are given in table 2.

Block size (S_B)	16 bytes (4 instructions)
Cache structure	direct-mapped
Cache size	[1Kb .. 64 Kb]

Table 2. Cache parameters

In addition, since we are only concerned with timing the cache behavior, we adopt a very simple timing model for instructions. An instruction is assumed to execute in $T_{hit} = 1$ processor cycles in case of a hit in the instruction cache or prefetch buffer, and in $T_{miss} = 10$ processor cycles otherwise.

3.2 Workload

The algorithms have been compared using 26 different synthetic task sets. Synthetic tasks are generated by an automatic tool. Input parameters to this tool are the size of the task, number of loops and nesting level, size and iterations of loops, number of if-then-else structures and their respective sizes. The user must provide the minimum and maximum desired values for these parameters. The tool randomly selects the actual value from this range. For the accomplished experiments, these parameters has been taken out from usual embedded workload, like Fast Fourier Transform, DSP algorithms for signal processing, sorting algorithms and matrix operations. However, some tasks has been created with no reference, but aimed to stress the locking cache architecture. The characteristics of tasks and task sets are summarized in table 3.

Number of tasks per task set	[3..8]
Maximum task set code size	64KB
Number of different tasks	50
Tasks code size	[1KB..32KB]

Table 3. Workload parameters

Within a task set, the task periods T_i, equal to their deadlines D_i have been adjusted such that the system is schedulable with both conventional and locking cache.

An experiment is defined by a pair (task set, cache size). 146 experiments have been conducted. Only experiments whose total code size is larger than the cache size have been considered in the following analysis, on the one hand because this is the most realistic situation and on the other hand because both algorithms would behave identically for task sets smaller than the cache size.

3.3 WCET and response time computation

Tasks worst-case execution times estimates (WCETs) are computed using a *tree-based* approach [15, 16]. The WCET estimate of a task is computed using recursive formulas that operate on the task's syntactic tree (a node in the tree represents a control structure – loop, conditional, sequence of blocks –, a leave represents a basic block – branch-free sequence of instructions). As we voluntarily ignore hardware components other than the instruction cache, the WCET estimate of a basic block BB can be computed in a straightforward manner from the WCET of its program lines pl: $WCET(BB) = \sum_{pl=1}^{npl} WCET(pl)$, with $WCET(pl) = T_{hit}$ or T_{miss} depending on whether program line pl has been locked in the instruction cache or not.

The response times of tasks are computed using CRTA (Cache-aware Response Time Analysis), which extends the well-known exact response time analysis (RTA) schedulability test [4, 18] to take cache-related preemption delays into account. Given a task τ_i, CRTA works by considering the interferences produced by the execution of the higher priority tasks on τ_i within an increasing time window w_i^n (n is the recurrence index). The response time R_i of task τ_i is the fixed point of the sequence given in equation 2 below, where $hp(\tau_i)$ denotes the set of tasks that have a higher priority than τ_i and γ denotes the cache-related preemption delay (the equations assume tasks with distinct priorities). In our context, the value of γ is the time needed to reload the prefetch buffer plus the related context switch penalty, and is a constant.

$$w_i^0 = C_i$$
$$w_i^{n+1} = C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil * (C_j + \gamma) \rightarrow R_i \quad (2)$$

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

3.4 Comparison metrics

From the cache contents generated by the two algorithms of section 2, we present a statistical analysis of the performance of the resulting task sets. We focus on *worst-case* performance metrics, computed from the tasks WCETs and the tasks response times:

- **Processor (worst-case) utilization (U).** The processor utilization ($\Sigma_{i=1}^{N} \frac{C_i^{'}}{T_i}$, where $C_i^{'}$ is the worst-case execution time of task τ_i including all cache effects) is an interesting metric because it allows to know a lower bound of the overall spare CPU capacity, which can be used for instance for executing soft real-time tasks. The lower the utilization, the better the cache contents selection algorithm.
- **Normalized (worst-case) slack (S).** The normalized slack for a task τ_i is defined as $S_i = \frac{D_i - R_i}{D_i} = 1 - \frac{R_i}{D_i}$, with R_i the task response time computed using response time analysis. S_i gives information about how close a task is from missing its deadline. The larger the normalized slack, the better the cache contents selection algorithm. This metric is interesting because it gives a task-level view of the spare processor capacity, whereas the utilization gives a system-level view of the system space capacity. This metric may be used to estimate which task has to (or may be) modified while keeping the system schedulable.
- **Execution time of the cache contents selection algorithms**. Although their speed is not the main concern since cache contents are selected off-line, a fast algorithm is better from the standpoint of development comfort and productivity.

4 Algorithm comparison

4.1 Comparison of utilization

Let U_{ri} (resp. U_{gi}) denote the (worst-case) utilization of an experiment i when using the reference-based (resp. genetic) cache contents selection algorithm. Since the lower the utilization the better the performance, values of $\Delta U_i = U_{ri} - U_{gi}$ below zero demonstrate the superiority of the reference-based algorithm, whereas values above zero demonstrate the superiority of the genetic algorithm. Values of ΔU_i are in the interval $]-1;+1[$.

Table 4 gives statistics on values of $\Delta U_i = U_{ri} - U_{gi}$. The average value of ΔU_i is very close to zero, whereas its median value is zero. The standard deviation and quartiles show that the vast majority of values are very close to zero. This is confirmed by Box and Whisker plot showed in Figure 1, where values into interquartile range are almost identical to the average and mean.

Number of experiments	146
Average	0.0125386
Median	0
Standard deviation	0.0539221
Minimum	-0.05984
Maximum	0.281344
Lower quartile	-0.0006595
Upper quartile	0.000055
Experiments with P<0	69 (47.3%)
Experiments with P=0	11 (7.5%)
Experiments with P>0	66 (45.2%)
95% Confidence interval for average	[0.0037184;0.0213588]
Stnd. skewness	17.7292
Stnd. kurtosis	30.1522

Table 4. Statistics summary for $\Delta U_i = U_{ri} - U_{gi}$

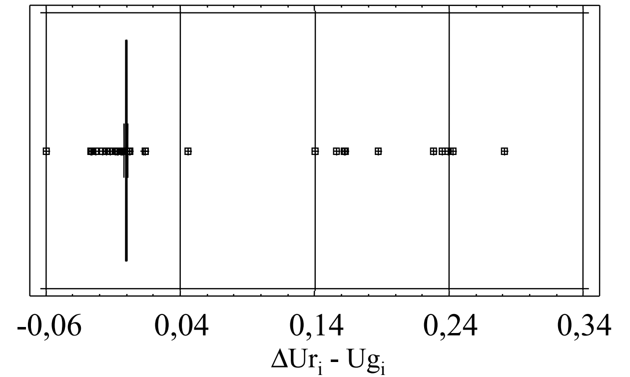

Figure 1. Box and Whisker plot for $\Delta U_i = U_{ri} - U_{gi}$

Confidence interval for average, that contains the average value, allows considering the average as the true average. However, there are two peculiarities. Firstly, the number of experiments with ΔU_i below zero is slightly greater

than the number of ΔU_i values over zero. Secondly, the minimum and maximum, the Box and Whisker plot, and the frequency histogram in Figure 2, show that ΔU_i values are greater when the genetic algorithm provides better performance.

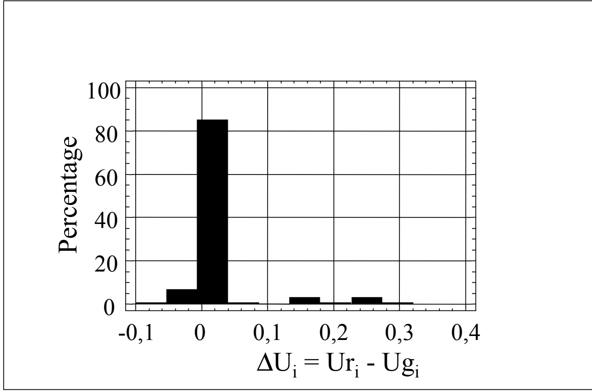

Figure 2. Frequency histogram (percentage of values of $\Delta U_i = U_{ri} - U_{gi}$ inside the range pointed by the x-axis)

In order to determine if these two peculiarities are significant, we have undertaken a deeper statistical analysis of the samples. More precisely, we have carried out t-test, signed test and rank signed test of null hypothesis with alpha equal to 5% [10]. While the first test rejects the null hypothesis, that is, the average is not zero and the genetic algorithm provides better utilization, the other two tests do not reject the hypothesis, so the mean is zero and both algorithms provides the same performance. Since the last two tests are less sensitive to outliers, the data does not come from a normal distribution as skewness and kurtosis show (values outside the range of -2 to +2 indicate it), and there is a large number of outliers in the Box and Whisker plot, the t-test is less reliable than the other two. Therefore, it can be assured that both algorithms statistically yield the same utilization.

4.2 Comparison of slack

An interesting comparison metric is the tasks (normalized) slack (simply called hereafter *slack*). Results concerning tasks slack are presented in two different manners:

- statistical study of the worst-case (smallest) slack in experiments. This metrics gives an idea of the system robustness: any perturbation whose duration is shorter than the worst-case slack can be supported by the system
- statistical study of the average slack of individual tasks

4.2.1 Worst-case slack per experiment

Let $WSri$ (respectively $WSgi$) denote the worst-case slack of tasks in experiment i when using the reference-based (respectively genetic) cache contents selection algorithm. A summary of the statistics of $\Delta WSi = WSri - WSgi$ is given in Table 5. Values of ΔWSi are in the interval]-1,+1[. Since the greater the slack, the better the algorithm, values of ΔWSi below zero indicate a better behavior of the genetic algorithm, while values over zero indicate a better behavior of the reference-based algorithm.

Number of experiments	146
Average	-0.0180381
Median	0
Standard deviation	0.0767092
Minimum	-0.458044
Maximum	0.067
Lower quartile	-0.00005
Upper quartile	0.00063667
Experiments with $\Delta WSi < 0$	63 (43.1%)
Experiments with $\Delta WSi = 0$	13 (8.9%)
Experiments with $\Delta WSi > 0$	70 (48.0%)
95% Confidence interval for average	[-0.0305857;-0.00549053]
Stnd. skewness	-20.3167
Stnd. kurtosis	42.5163

Table 5. Statistics summary for $\Delta WSi = WSri - WSgi$

The statistics summarized in Table 5 give contradictory information: average and minimum value points to larger slack when using genetic algorithm, while interquartile range and frequencies points to larger slack when using reference-based algorithm. Other statistics, like median or standard deviation do not help to deciding if there exists any difference between algorithms. From the three null hypothesis test (t-test, sign test and signed rank test), the first of them rejects the null hypothesis, but the last two ones does not reject the null hypothesis. Therefore, we can conclude that there is no statistical difference between the two algorithms. Both algorithms thus statistically exhibit the same robustness.

4.2.2 Slack of individual tasks

We now consider the distribution of the slacks of individual tasks. Table 6 summarizes the statistics concerning $\Delta S_i = S_{ri} - S_{gi}$, where S_{ri} (resp. S_{gi}) is the slack of task τ_i with the reference-based (resp. genetic) algorithm. Values of ΔS_i below zero point to a better behavior of the genetic algorithm, whereas values over zero point to a better behavior of the reference-based algorithm.

Number of tasks	610
Average	-0.0107234
Median	-0.000015
Standard deviation	0.0471117
Minimum	-0.458044
Maximum	0.067
Lower quartile	-0.002376
Upper quartile	0
Tasks with ST<0	372 (61%)
Tasks with ST=0	103 (16.9%)
Tasks with ST>0	135 (22.1%)
95% Confidence interval for avg.	[-0.014462; -0.00698473]
Stnd. skewness	-60.8165
Stnd. kurtosis	211.039

Table 6. Summary statistics for $\Delta S_i = S_{ri} - S_{gi}$

In the table, the average and median of ΔS_i show negative values, pointing a better behavior when the genetic algorithm is used to select cache contents. Besides, minimum, maximum, interquartile and confidence interval, show that values of ΔS_i are skewed towards negative values. All of these basic statistics, as well as a deeper statistical analysis (t-test, signed test and signed rank test for null hypothesis with alpha 5%), show that the slacks of individual tasks with both algorithms are statistically different, and that the genetic algorithm exhibits a better behavior than the reference-based algorithm.

However, although these differences are statistically significant, the slack improvement provided by the genetic algorithm is small (maximum improvement of 7%, average improvement of 1%).

Better behavior of the genetic algorithm compared to the reference-based algorithm is due to the fitness function used by the genetic algorithm. The fitness function is based of the response time of tasks, which is a more precise, albeit longer to compute, metric than the overall utilization used by the reference-based algorithm.

4.3 Impact of system parameters

We have studied the impact on several parameters on the utilization and slack. Studied parameters were: (i) task code size (individual task code size and cumulated code size of the task sets); (ii) number of tasks in the task sets; (iii) ratio between code size and task size; (iv) tasks priorities (v) number of executed instructions per-task (length of execution paths).

A statistical study did not show any significant impact of the first fourth parameters on the compared behavior of the genetic and reference-based algorithm, both considering utilization and slack.

Among the studied parameters, the only factor having a slight impact is the number of executed instructions per task. For tasks with the longest execution paths (from 6 million to 12 million of instructions), corresponding to loop-intensive programs, the genetic algorithm yields slightly better results than the reference-based algorithm, both considering the system utilization and slack. This phenomenon comes from the way cache contents is selected in the reference-based algorithm. The reference-based algorithm bases its selection of cache contents on the references along *single* path, the worst-case execution path (WCEP); considering only that path may cause the WCEP to change once the cache contents is selected. Although this phenomenon may appear for any program[1], its impact is quantitatively greater when the portion of code causing the WCEP to change is repeated many times.

4.4 Execution time of algorithms

Blocks to be locked in cache are selected during design phase, and thus the speed of cache contents selection does not affect the performance of the actual system. However, the speed of the algorithm for cache contents selection may be important if source code of the system tasks are modified frequently.

The execution times of both algorithms have been measured on a Pentium II, 200Mhz processor running Linux. Both algorithms have been implemented in C. Over the 146 experiments we carried out, the reference-based algorithm always executes in less than two minutes. In comparison, the genetic algorithm took between two and six hours to execute for most experiments, and in some cases more than ten hours. Obviously, the reference-based algorithm is extremely faster than the genetic algorithm.

5 Concluding remarks

We have compared in this paper the behavior of two algorithms for static global instruction cache locking: one using a *genetic algorithm* for cache contents selection [3] and a pragmatical algorithm, called *reference-based* algorithm [14], which uses the string of memory references issued by a task on its worst-case execution path as an input of the cache contents selection algorithm. Experimental results have shown a similar behavior of the compared algorithms, in terms of both processor utilization and worst-case slack per task set. The genetic algorithm has demonstrated a slightly better behavior regarding the average slack of individual tasks. However, its large execution time may not in

[1] It appears when the timing difference between the WCEP and the other paths in the program is low, and was shown not to appear too often in practice, as shown in [13].

general be worth paying such a small improvement over the reference-based algorithm. One could envision using both algorithms jointly, for instance by using the cache contents produced by the reference-based algorithm, as a base cache contents for the genetic algorithm.

Further work would be required to compare the performance of the algorithms studied in this paper with the work of Vera. et al [19], based on a more selective use of cache locking.

6 Acknowledgments

This work has been supported in part by the Spanish *Comision interministerial de Ciencia y Tecnologia* under project CICYTTIC2003- 08106-C02-01

References

[1] 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 *Proc. of Tenth International Workshop on Hardware/Software Codesign (CODES 2002)*, Estes Park, Colorado, May 2002.

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

[3] A. M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Using genetic algorithms in content selection for locking-caches. In *Proc. of the IASTED International Symposium on Applied Informatics*, pages 271–276, Innsbruck, Austria, Feb. 2001.

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

[5] 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.

[6] Y.-H. Lee, D. Kim, M. Younis, J. Zhou, and J. McElroy. Resource scheduling in dependable integrated modular avionics. In *Proc. of the 2000 International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, pages 14–23, New York, USA, June 2000.

[7] J. Leung and J. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

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

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

[10] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

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

[12] E. Petrank and D. Rawitz. The harness of cache conscious data placement. In *Proc. of 29th ACM SIGPLAN-SIGACT Simposium on Principles of Programming Languages*, pages 101–102, Portland, Oregon, 2002.

[13] 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.

[14] 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.

[15] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sept. 1989.

[16] A. C. Shaw. Reasoning about time in higher-level langage software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[17] E. Tamura, J. V. Busquets-Mataix, J.J. Serrano, and A. Marti Campoy. A comparison of three genetic algorithms for locking-cache contents selection in Real-Time systems. In *Proceedings of the 7th International Conference on Adaptive and Natural Computing Algorithms (ICANNGA05)*, pages 462–465, Coimbra, Portugal, March 200

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

[19] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS03)*, Cancun, Mexico, 2003.