Using dynamic, full cache locking and genetic algorithms for cache size minimization in multitasking, preemptive, real-time systems

Antonio Martí Campoy, Francisco Rodríguez-Ballester, and Rafael Ors Carot

Universitat Politècnica de València, Departamento de Informática de Sistemas y Computadores, 46022 València SPAIN

Abstract. Cache locking have shown during the last years their usefulness easing the schedulability analysis of multitasking, preemptive, real-time systems. Cache locking provides a high degree of predictability while system performance is maintained at a similar level to that provided by regular, highly unpredictable, non-locked cache. Cache locking may also be useful to reduce hardware costs by means of reducing the size of the cache memory needed to make a real-time system schedulable. This work shows how full, dynamic cache locking may help to reduce the size of the cache memory versus a regular cache. This reduction is possible thanks to a genetic algorithm that selects the set of instructions that have to be locked in cache to provide the maximum cache size minimization while keeping the system schedulable.

Keywords: Genetic algorithm, Real-Time Systems, Cache Locking, schedulability analysis, cost-saving

1 Introduction

Cache memories are an important advance in computer architecture, providing significant performance improvements. However, in the area of real-time systems, the use of cache memories introduces serious problems regarding predictability. The dynamic and adaptive behaviour of a cache memory reduces the average access time to main memory, but presents a non-deterministic fetching time [7]. In multitasking, preemptive, real-time systems, estimating the Worst Case Response Time (WCRT) of every task in a system populated with a regular cache becomes a problem with a solution hard to find due to the interference on the cache contents produced among the tasks.

In recent years, the use of locked caches have appeared as a solution to ease the schedulability analysis of multitasking, preemptive, real-time systems maintaining, at the same time, similar performance improvements than systems populated with regular cache memories. Several works have been presented to apply cache locking in real-time systems, both for instructions [3][15][14][1] and data [16]. In this work, we focus on instruction caches only, because 75% of accesses to main memory are to fetch instructions [7].

The paper is organised as follows: section two describes the objective and the goal of this work. Section three brief introduces the use of cache locking in multitasking, real-time systems. Section four describes with detail the Genetic Algorithm that selects the contents to load and lock in cache. Sections five, six and seven show the setup, the procedures and the results of experiments carried out. Finally, conclusions are presented in section eight.

2 Rationale

The ability to implement custom processors in FPGAs or ASICs has partly simplified the choice of the processor for a real-time system [11]. With these it is possible to build a system with its performance tailored to the actual requirements, decreasing the cost of the resulting system. What's more important, the designer can incorporate those architectural improvements that interest he most.

Although achieving predictability and easing the schedulability analysis is the main goal and advantage of cache locking, it may help to minimize the cache size thus saving costs and reducing the power consumption of the system.

The goal of this work is to show that LSM-dynamic use of cache locking may reduce the cache size needed to make a real-time system schedulable in front of a regular cache. And this advantage is added to the predictability and easiness of analysis provided by cache locking.

3 Use of cache locking

Cache locking means to keep all or part of cache contents away from replacement policy, or to allow replacement only in particular scenarios. In brief, the adaptive behaviour of regular caches is reduced or completely eliminated.

When the entire cache is locked, it is called full locking. If only some parts of the cache contents are locked, no matter if these parts are cache ways or cache lines, it is called partial locking. [6]

In multitasking systems, when locked contents remain unchanged for the whole life of the system, it is called static use or static cache locking.

If locked contents may change in a controlled way, for example in task switch, it is called dynamic use of cache locking. Some authors prefer to call this multiplexed or time-shared cache.

Several previous works in the literature apply cache locking over individual, isolated tasks in order to improve or accurately estimate the Worst-Case Execution Time (WCET) of a task. The work developed in this paper deals with multitasking, preemptive, real-time systems that presents a more complex problem, because includes the WCET estimation together with the WCRT estimation, dealing with intra-task and inter-task interference related to cache [2][8].

3.1 Full, dynamic use of cache locking

In here called dynamic use of cache locking every task may use and lock the cache in full for its own instructions. Replacement of cache contents is allowed,

but at some moments only.To our best knowledge, the first proposal of dynamic use of cache locking appears in [3] where cache contents may change only when a task starts its execution or resumes after a preemption, flushing the cache contents and reloading it with its own set of selected instructions. This is the behaviour considered in this work.

WCET estimation for each task in the system is easy, because during the execution of the task the cache contents is always the same, even after a preemption, because the first action when a task resumes its execution is to reload the cache with its own instructions.

Cache Response Time Analysis (CRTA) [2] is here used to estimate the response time of tasks, taking into account the time needed to reload the cache before a task runs or resumes its execution. This time, called cache refill penalty or cache-related preemption delay is added to the execution time of each task for each run and for each preemption, giving a safe upper bound of its WCRT.

Reloading cache contents when a task starts or resumes execution may be accomplished by software or hardware means. By software, the scheduler is overloaded with a routine that read from main memory the list of main memory blocks (instructions) that were off-line selected to load and lock in cache. For each main memory block, the routine accesses main memory two times: first, to fetch the address of the block to be loaded, and second, to effectively read this block and transfer it to the cache memory by means of specific instructions. Some other constant overload applies, and the total temporal cost to run this routine is added to CRTA as the cache refill penalty.

Loading instructions by means of software adds an important overhead, so in [15] a hardware solution is proposed. An extra, dedicated memory is added, called the Locking State Memory (LSM). Its role is to store the status of every main memory block, that is, whether it is selected or not to be locked in cache. This way the LSM provides a mechanism to discriminate which blocks must be loaded into the cache and hence a way to allow for automatic, on-demand loading of the selected main memory blocks. In other words, instead of locking the selected blocks into a locked cache, the same effect can be attained by avoiding loading into the cache the unselected blocks.

4 Using a Genetic Algorithm to select cache contents

Cache locking intrinsically provides predictability. But other aspects, like performance, depend on the set of instructions selected to be locked in cache. Random selection of these instructions will give predictability and easiness of analysis, but system performance may be seriously degraded.

Several algorithms [5] have been developed in order to select instructions to load and lock in cache that improves system performance, or at least, maintains the system performance close to a system populated with a regular, non-predictable cache memory.

In [9] a new version of a genetic algorithm is proposed. The target of this algorithm is not to improve system performance but to reduce the size of the

cache memory needed to get an schedulable real-time system. This algorithm is used on static locking and outperforms previous results [4] where a performance-targeted algorithm was used.

In this paper the algorithm is adapted to be used on dynamic locking. This adaptation is done primarily in two points: in the representation of the problem in order to allow each task to fully utilize the cache, and in the evaluation of individuals where the CRTA equation corresponding to the dynamic cache locking using an LSM is used. The GA is executed in the design phase of the real-time system, so its execution time does not affect nor results in overhead for the real-time system. Below the main operators of the GA are described.

Problem Representation. A tri-dimensional matrix stores the status of main memory blocks in order to determine whether they are selected or not to be loaded and locked into cache. The first index used to access this matrix identifies to which task the memory block belongs, the second index identifies the set or cache memory line in which the block is mapped, and the third dimension is used to store the list of blocks that are mapped to that cache set/line together with their corresponding status. This rather complex structure allows crossing two individuals and mutating the resulting individuals guaranteeing that the cache mapping function will not be violated irrespective of the cache associativity degree. In fact, the representation is a list (of tasks) of lists (of cache sets/lines) of lists (of selected blocks for each cache set/line). The last list (selected blocks) is the responsible of fitting the mapping function. When individuals are created, the number of selected blocks for each set/line is at most the cache associativity degree. In crossover, the splitting point is restricted to the second dimension, avoiding the division of the third dimension of an individual, so the number of selected blocks remain unchanged and in a valid range. Regarding mutation, it only unlocks blocks, so it is not possible that the resulting number of selected blocks will be over the associativity degree.

Initial Population. The initial individuals are created by selecting a pseudo-random number of blocks to load and lock in cache. These blocks may belong to any task. Also, initialization creates a single individual with all of the blocks of every task marked as selected for being loaded and locked in cache memory. The purpose of including the special individual is to broaden the initial search space and guarantee that the algorithm starts with at least one schedulable individual.

The fitness function results from the combination of the result of the schedulability test, the number of used cache lines, and the system global utilisation. This way, the fitness value for an individual is a 3-tuple (S, L, U), where S is the boolean result from the schedulability test, L is the cache size, measured in cache lines, the individual needs, and U is the global utilisation of the system. An individual i with fitness (S_i, L_i, U_i) is better than individual j with fitness (S_j, L_j, U_j) if:

$$
\begin{aligned}
&i)\ S_i \textit{ and } not(S_j) \textit{ or} \\
&ii)\ S_i \textit{ and } S_j \textit{ and } (L_i < L_j) \textit{ or} \\
&iii)\ S_i \textit{ and } S_j \textit{ and } (L_i = L_j) \textit{ and } (U_i < U_j)
\end{aligned}
$$

That is, and individual is better than other if it is schedulable and the other one is not, or if it uses a smaller cache (in case both are achedulable), or, in case of a tie, it presents a lower system utilisation.

As selection policy binary tournament has been chosen. To select a parent two individuals are pseudo-randomly chosen, their fitness functions compared, and the better individual is chosen. The same procedure is used to choose the other parent. Besides this selection, an elitist selection is introduced. Two copies of the best individual are made, one of the copies is exposed to mutation while the other one suffers no mutation and is effectively incorporated into the next generation without any modification.

Single point crossover is used in this algorithm. The single point crossover is obtained by splitting every parent at a locus given by a pair $(task\_number, set\_number)$ and creating two new individuals by exchanging the parts of the two parents.

In the mutation process the algorithm pseudo-randomly selects a set of locked blocks and marks them as unlocked, thus decreasing the number of locked blocks and therefore, the cache size.

The GA execution finishes when a predetermined number of generations has been reached. This value and the remaining input parameters are detailed below: Population size: 200 individuals; Termination condition: 5000 generations; Crossover probability: 100%; Mutation probability: 8%; Runs for each GA experiment to avoid seed effects: 25. The average cache size of these runs is presented in this work.

The input parameters of the genetic algorithm are: Algebraic expressions for tasks' WCET estimation [3]; List of main memory blocks used by each task; Periods of tasks; Hit and miss times; Cache line / main memory block size; Cache mapping function.

The output of the GA includes: worst-case execution time and worst-case response time of each task, the minimum cache size that makes the system schedulable, and the list of main memory blocks selected to be loaded and locked in the cache.

It is important to recall that execution time and response time of tasks provided by the genetic algorithm are not simulated, but estimated by means of cache-aware analysis methods (CRTA).

5 Experiment setup

In order to study the ability to reduce the cache size the real-time systems presented in [9] were used. In this set of systems there are 14 different task sets. Tasks are artificially created to stress the locking/regular cache scheme. Main parameters of each task are defined, like its size, the number and size of loops and their nesting level, number of if-then-else structures and their respective sizes. These parameters are fixed or randomly selected. Then, a simple software tool creates the assembly code. This code is MIPSR2000 compatible.

Table 1 shows the different sets of tasks, the number of tasks per set, the sum of task sizes, their average size, and the system utilisation when the system runs on a 4096 lines-size regular cache.

Task periods are manually adjusted to force different number of preemptions among the tasks and therefore setting the system utilisation in different values, creating four scenarios. In all four scenarios and for all task sets, task deadlines are equal to task's periods and priority is assigned by the Rate Monotonic policy (the shorter the period the higher the priority).

The tasks in the first scenario have large periods, much longer than their respective response times, so there are no preemptions among tasks and the resulting system utilisation is low: around 30% when the system runs on a regular cache larger than the sum of all tasks sizes, that is, when only compulsory misses may occur. This scenario is called Low Interference (LI).

The tasks in the second scenario have also large periods, but not very far from their response times, so some preemptions may occur. System utilisation is this case is around 60% when system runs on a regular cache larger than the sum of all tasks sizes, that is, when only compulsory misses may occur. This scenario is called Low Medium Interference (LMI).

In the third scenario the tasks have periods close to their respective response times so many preemptions occur, increasing response times and therefore, system utilisation. In this case the system utilisation is around 80% when system runs on a regular cache larger than the sum of all tasks sizes, that is, with compulsory misses only. This scenario is called Medium High Interference (MHI).

Task periods in the fourth scenario are very close to their response times, so the number of preemptions is large and the system utilisation grows up to 95% when system runs on a regular cache larger than the sum of all tasks sizes. This scenario is called High Interference (HI).

Table 1 shows the system utilisation for each set of tasks, and for each configuration of deadlines when the system runs on a regular cache larger than the system size.

A total of 56 systems = 14 (task sets) x 4 (periods sets) are evaluated under two architectures: regular caches, and full, dynamic cache locking by means of hardware LSM.

Using the same task set but different task periods allows assessing the behaviour of the architectures and algorithms in front of a wider range of cache size requirements: as the task interference increases a larger cache is necessary to keep the system schedulable.

Results for regular caches come from simulation, while results for LSM cache locking come from estimation of WCET and WCRT by means of CRTA. This way, actual response time of tasks when using regular caches may be longer, but never shorter. In the other hand, actual response time of tasks when using cache locking may be shorter, but never longer, because they are estimated using conservative approaches. Conventional cache benefit from this situation because optimistic values are used for regular cache in front of conservative values for cache locking.

Table 1. Main parameters of the set of tasks used in experiments, and system utilisation for each scenario

Task set	# of tasks	Total size (instructions)	Average size (instructions)	LI (%)	LMI (%)	MHI (%)	HI (%)
1	3	2565	855.0	34.8	62.2	78.7	96.1
2	4	3397	849.3	35.0	60.6	78.1	98.4
3	8	1640	205.0	43.9	61.5	79.0	93.1
4	8	1640	205.0	36.3	59.5	82.3	95.7
5	5	2124	424.8	38.0	58.6	76.1	97.5
6	4	3252	813.0	32.0	62.4	76.7	93.9
7	3	3678	1226.0	40.9	57.1	80.5	97.8
8	3	1923	641.0	34.5	61.2	78.0	91.5
9	5	3086	617.2	38.7	58.8	78.9	96.5
10	3	3602	1200.7	35.0	62.0	80.2	93.2
11	4	1904	476.0	32.4	61.7	81.8	95.2
12	3	2378	792.7	36.5	68.7	83.3	96.8
13	5	790	158.0	29.4	59.3	75.9	98.0
14	5	2146	429.2	28.2	60.8	82.5	92.0

Regarding cache characteristics, fetching an instruction from cache takes 1 cycle while fetching an instruction from main memory takes 10 cycles. The only mapping function considered for cache locking is direct mapping, because this one is the most restrictive and provides the worst performance for cache locking [3][13]. The maximum size of the cache is 4096 lines, where each cache line contains four words, and each word is four bytes long. The instruction size is four bytes also. A main memory block presents the same size than a cache line.

The system includes a prefetch buffer with the size of one cache-line in order to get advantage of spatial locality when the processor fetches a main-memory block not locked in cache.

6 Experiment procedures

The goal of the experiments carried out is to find the minimum cache size required to keep the system schedulable. This minimum size depends on the characteristics of the software system (number and size of tasks, periods, deadlines) and the underlying architecture the system runs on (regular cache or LSM-based dynamic cache locking).

For regular cache, in order to find the minimum cache size needed to make the system schedulable, several simulations of the system execution have been accomplished varying the cache size, starting in 1 line and increasing one by one, finishing when the cache size is one line larger than the total sum of the sizes of the system tasks. It would be possible to stop the experiments when system becomes schedulable but we want to study if there are discontinuities in the size of the cache that makes the system schedulable. No discontinuities were found.

Simulations were accomplished using a modified version of SPIM, a MIPSR2000 simulator [12]. Input parameters for simulation are: Code of tasks in assembly; Periods of tasks; Cache size; Mapping policy (direct, two-way and full associative were tested, choosing the best result for each system, that is, the smaller cache size, usually direct mapping); Hit and miss times (1 and 10 cycles); Cache line / main memory block size (four four-byte-size instructions).

The output from simulation includes: worst execution time of tasks, worst response time of tasks, system utilisation, and schedulability test result (yes/no).

For cache locking the minimum cache size is found by a direct search using the genetic algorithm previously described. The genetic algorithm uses cache-aware analysis methods to estimate the response time of task and decide if the system is schedulable for a particular cache size.

The experiment setup and procedures give us a total of 112 experiments (56 systems multiplied by two cache architectures under study).

7 Experiment results

The first outcome from experiment results is completely unexpected. For 14 experiments, the genetic algorithm considering the LSM architecture is unable to find a cache size to make the system schedulable. Even using a cache larger than the sum of the sizes of all the tasks that belong to the system, the genetic is unable to find a solution. The 14 experiments with no solution belong to the set of systems with the highest degree of task interference and very high system utilisation (scenario HI).

But the systems are in fact schedulable because when using a regular cache, there is a cache size for each system that makes it schedulable. So the problem is not in the real-time systems.

The reason the GA fails in finding a schedulable-making cache size for the LSM architecture is because of an excessively conservative approach when estimating the cache refill penalty.

The proposed CRTA analysis that is embedded in the GA considers that in every preemption the cache is completely flushed, and when a task resumes its execution, the full set of instructions selected to lock in cache is reloaded, without regard about how many of these instructions are useful, that is, will be executed before the next preemption.

Actually, both for regular and LSM cache, after a preemption, only instructions that are fetched by the processor between two preemptions may produce a miss, increasing the cache refill penalty. The instructions executed between two preemptions may be all of those selected to be locked in cache, or only a small subset, depending on the task structure and the time between preemptions.

With the conservative analysis the cache refill penalty is well-known, constant, and safe. But may be excessively larger. And the larger the number of preemptions, the worse the overestimation. This is the reason this problem arise for the systems with a high degree of interference; tasks suffer a lot of preemptions.

Table 2. Main statistics for LREG - LLSM

Statistic	LI	LMI	MHI	HI
Average	+123.35	-37.71	-146,5	-951.0
Min	-59	-414	-1092,0	-2074
Max	+327	+76	+189	-54
Cases<0	2	5	10	14
Cases>0	12	9	4	0
Avg. Reduction	47%	-5%	-14%	-39%
P-value t-test	0.0039	0.387	0,118	0.00059
T-test answer to null hypothesis	Reject	Do not reject	Do not reject	Reject
P-value Wilcoxon test	0.0057	0.66	0,09	0.00109
Wilcoxon test answer to null hypothesis	Reject	Do not reject	Do not reject	Reject

In following results, for the 14 systems where the GA is not able to find a cache size, we will use a cache with as many lines as the sum of the sizes of all tasks of the system, and manually disables the locking mechanism. All instructions will be loaded in cache when they are fetched by the processor, and they will remain in cache because of the size of cache makes no conflicts.

Analysis of result is accomplished grouping systems in the four previously described scenarios regarding its degree of task interference and value of utilisation (LI, LMI, MHI and HI). For each one of the four groups, a paired sample analysis is presented [10].

Table 2 shows the main statistics for LREG (number of cache lines used by the regular cache) minus LLSM (number of cache lines used by the LSM cache locking). for the four scenarios. The table also contains the p-value of a t-test and a Wilcoxon signed-rank test of null hypothesis and the answer to these tests, using an $\alpha = 0.05$.

Figure 1 shows the ratio LREG to LLSM for systems grouped in the four scenarios. The ratio is calculated as $(LREG/LLSM) - 1$. This figure shows if the LSM needs a smaller cache than the regular cache and the relative magnitude of cache saving. Values over 0 means the LSM needs less lines of cache than a regular cache. In the other hand, values below indicates that regular cache beats LSM, using a smaller cache to make schedulable a system.

In Figure 1-a) can be clearly observed that the LSM is able to make the systems schedulable using a smaller cache in most of the cases when the degree of interference between tasks is low (LI); this is confirmed by the answer of both tests which rejects the Null Hypothesis, meaning there is a statistically significant difference in the cache size needed by the LSM and the regular cache. The cache saving by means of using the LSM is in average of 45%, but in almost half of the cases the cache size reduction is over 100%.

In Figure 1-b), systems in scenario LMI, can be seen that there are more cases where the LSM needs a smaller cache, but in the cases where the LSM

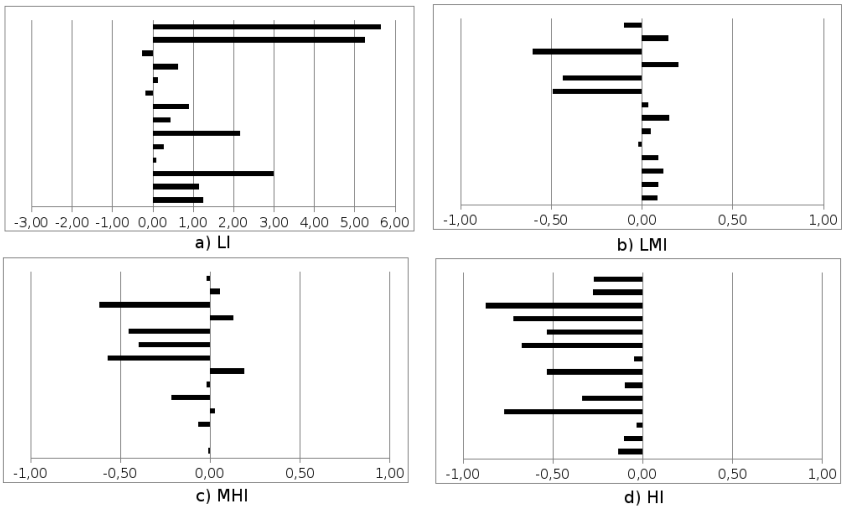

Fig. 1. Percentage of reduction of cache using LSM for experiments with periods a)LI, b)LMI, c)MHI and d)HI $(LREG/LLSM) - 1$.

needs a bigger cache the cache size is much larger than the cache size needed by regular use. Attending the statistics in the third column (LMI) of table 2, the average cache size reduction is about -5%, that is, the LSM needs in average larger caches than the regular use of cache, but the answer of both tests is that the Null Hypothesis cannot be rejected; that is, there is no statistically significant differences between the cache sizes used by the two studied uses of cache.

In Figure 1-c), systems in scenario MHI, a slight bias to negative values can be appreciated; that is, there are more cases where the LSM needs a larger cache to make the system schedulable. Attending the statistics shown in the fourth column (MHI) of table 2, the LSM needs, in average, a 10% larger cache than the regular use of cache (reduction of -0.09%), but response of Null Hypothesis tests are do not reject; that is, there is no statistically significant difference between the cache size used by the two studied uses of cache.

Finally, the figure 1-d) shows the ratio LREG to LLSM for systems with High Interference, HI, calculated as $(LREG/LLSM) - 1$. The data in this figure is very clear: for all cases using a regular cache needs a smaller cache to make the system schedulable. And in front of LSM, the cache needed by the regular cache is about 30% smaller in average, and even more than 50% smaller in some cases. Statistics on the fifth column of table 2 confirms this conclusion, in particular the tests that reject the Null Hypothesis; that is, there are statistically significant differences between the regular use of cache and the LSM.

Results of the analysis of the four groups are coherent with the analysis of overestimation described in the beginning of this section. The larger the number of preemptions, the larger the overestimation. This way, the larger the interference between tasks, the smaller the cache saving by means of LSM dynamic use

of cache locking. For one of the scenarios (LI) the LSM is able to significantly reduce the size of cache required to make the system schedulable. That is, using cache locking by means of a LSM architecture helps the system designer to save costs and power. In the other extreme, another scenario (HI) shows that the LSM needs larger caches than the regular use, so its use is not recommended if the main concern is related to system costs or power consumption. For the two intermediate scenarios (LMI and MHI) there is a tie, that is, both the LSM and the regular cache will need the same size of cache to make the system schedulable. So, in general terms and talking about these intermediate scenarios, there is no significant difference in using a regular cache or a dynamically locked cache, concerning about system cost and size only.

But cache locking adds a precious value to the design process of a real-time system because they provide predictability and easiness of analysis.

8 Conclusions

This work presents a new application of dynamic cache locking, extending the research from a previous proposal where static cache locking was applied to reduce the instruction cache size in preemptive, multi-task, real-time systems. Results from experiments carried out show, first, that for some systems the dynamic use of cache locking is not suitable due to the overly conservative considerations that have to be used in the schedulability analysis, so for high interference, high utilisation systems, regular caches provide better results.

Second, there is a large number of cases where no statistically significant difference exists, in average, in the cache size needed to make a system schedulable when using a regular cache or a locked cache. In these cases, and for the same cache size, cache locking offers, in front of a regular cache, the benefit of high predictability and simplicity in the schedulability analysis.

Only when the system utilisation is low a LSM-based locked cache allows a notable significant cache size minimization maintaining the predictable behaviour and ease of analysis.

Experiment results also show that for some systems, and under some conditions, there may be important differences in the cache size needed to make the system schedulable. Future work will study the system characteristics and conditions that determine the architecture that gets a smaller cache, so recommendations to the system designer can be done.

Acknowledgments

This work is partially supported by PAID-06-11/2055 of Universitat Politècnica de València and TIN2011-28435-C03-01 of Ministerio de Ciencia e Innovación

References

1. Aparicio, L.C., Segarra, J., Rodrguez, C., Vials, V.: Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-

time systems. Journal of Systems Architecture 57(7), 695 – 706 (2011), http://www.sciencedirect.com/science/article/pii/S1383762110001086, special Issue on Worst-Case Execution-Time Analysis
2. Busquets-Mataix, J., Serrano, J., Ors, R., Gil, P., Wellings, A.: Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In: Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE. pp. 204–212 (jun 1996)
3. Campoy, A.M., Ivars, A.P., Mataix, J.V.B.: Dynamic use of locking caches in multitask, preemptive real-time systems. In: Proceedings of the 15th World Congress of the International Federation of Automatic Control (2002)
4. Campoy, A.M., Rodríguez-Ballester, F., Ors, R., Serrano, J.: Saving cache memory using a locking cache in real-time systems. In: Proceedings of the 2009 International Conference on Computer Design. pp. 184–189 (jul 2009)
5. Campoy, A.M., Puaut, I., Ivars, A.P., Mataix, J.V.B.: Cache contents selection for statically-locked instruction caches: An algorithm comparison. In: Proceedings of the 17th Euromicro Conference on Real-Time Systems. pp. 49–56. IEEE Computer Society, Washington, DC, USA (2005), http://portal.acm.org/citation.cfm?id=1084012.1084148
6. Ding, H., Liang, Y., Mitra, T.: Wcet-centric partial instruction cache locking. In: Proceedings of the 49th Annual Design Automation Conference. pp. 412–420. DAC '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2228360.2228434
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 4th Edition. Morgan Kaufmann, 4 edn. (2006)
8. Lee, C.G., Lee, K., Hahn, J., Seo, Y.M., Min, S.L., Ha, R., Hong, S., Park, C.Y., Lee, M., Kim, C.S.: Bounding cache-related preemption delay for real-time systems. IEEE Transactions on Software Engineering 27(9), 805–826 (2001)
9. Martí-Campoy, A., Rodríguez-Ballester, F., Tamura Morimitsu, E., Ors, R.: An algorithm for deciding minimal cache sizes in real-time systems. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation. pp. 1163–1170. GECCO '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2001576.2001733
10. McPherson, G.: Applying and Interpreting Statistics. A Comprehensive Guide. Springer Texts in Statistics, Springer, 2nd edn. (2001)
11. Navabi, Z.: Embedded Core Design with FPGAs. McGraw-Hill Professional (2006)
12. Patterson, D., Hennessy, J.: Computer Organization and Design: The Hardware/software Interface. Morgan Kaufmann (1994)
13. Puaut, I., Pais, C.: Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In: Proceedings of the conference on Design, automation and test in Europe. pp. 1484–1489. DATE '07, EDA Consortium, San Jose, CA, USA (2007), http://dl.acm.org/citation.cfm?id=1266366.1266692
14. Sascha Plazar, J.C.K., Marwedel, P.: Wcet-aware static locking of instruction caches. In: Proceedings of the 2012 International Symposium on Code Generation and Optimization. pp. 44–52 (2012)
15. Tamura, E., Busquets-Mataix, J., Campoy, A.M.: Towards predictable, high-performance memory hierarchies in fixed-priority preemptive multitasking real-time systems. In: Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS-2007). pp. 75–84 (2007)
16. Vera, X., Lisper, B., Xue, J.: Data cache locking for tight timing calculations. ACM Trans. Embed. Comput. Syst. 7(1), 4:1–4:38 (Dec 2007), http://doi.acm.org/10.1145/1324969.1324973