The 9th International Conference on Real-Time and
Embedded Computing Systems and Applications.
Tainan, TW February 2003
Lecture Notes in Computer Science
Pages 328-341, vol. 2968, 2004

Schedulability analisys in EDF scheduler with cache memories*

A. Martí Campoy, S. Sáez, A. Perles, and J.V. Busquets

Departamento de Informática de Sistemas y Computadores,
Universidad Politécnica de Valencia,
46022, Valencia (SPAIN)
{amarti,ssaez,aperles,vbusque}@disca.upv.es

Abstract. Cache memories can improve computer performance, but its unpredictable behaviour makes difficult to use them in hard real-time systems. Classical analysis techniques are not sufficient to accomplish schedulability analysis, and new hardware resources or complex analysis algorithms are needed. This work presents a comprehensive method to obtain predictability on the use of caches in real-time systems using an EDF scheduler. Reaching a predictable cache, schedulability analysis can be accomplished in a simple way through conventional algorithms.
At the moment, this is the first approach to consider cache in this kind of scheduler. The method is based in the use of locking caches and genetic algorithms. Locking caches allows to load and lock cache contents, ensuring its remains unchanged. Genetic algorithms help to select the cache content that offers the best performance. Experimental results indicate that this scheme is fully predictable, and this predictability is reached with no performance loss for around 60% of cases.

1 Introduction

Modern microprocessors include cache memories in its memory hierarchy to increase system performance. General-purpose systems can benefit from this architectural improvement, because it tries to make efficient the average case. But hard real-time systems require the worst case to be bounded, and therefore, to take advantage of cache memories, they need special hardware resources and/or speficic system analysis that guarantee the timeliness execution of the code.

Basically, two problems arise when cache memories are used in multitask, preemptive real-time systems: intra-task interference, in the domain of a single task; and inter-task interference, in the domain of multitask systems. The former one makes harder to calculate the Worst Case Execution Time (WCET), because a task can replace its own instructions in cache due to conflict and capacity problems. When previously replaced instructions are executed again, a cache miss increases the execution time of the task. This kind of interference has to be taken into account in the WCET of each task.

* This work was supported by the Spanish Government Research Office (CICYT) under grants TAP99-0443-C05-02 and TIC99-1043-C03-02

The second problem is to calculate the cache-related preemption delay. This delay, also called inter-task or extrinsic interference, arises in preemptive multitasking systems when a task displaces from cache the working set of any other task of the system. When the preempted task resumes execution, a burst of cache misses increase its execution time over the precalculated WCET. This additional delay must be considered in the schedulability analysis.

The cache problems considered in this work deal with the resolution of cache interference in multitask, preemptive real-time systems. The paper only addresses the instruction cache problem and how it influences in the calculation of WCET of each task and in the schedulability analysis of the whole system, without regard to other architecture improvements.

Several solutions have been proposed to use cache memories in real-time systems. Some works analyse the cache behaviour to estimate the task execution time considering only the intra-task interference [1–4]. In [5, 6] the cache behaviour is analysed to estimate the task response time, but considering only the inter-task interference and using a precalculated cached WCET. These works deals only with fixed priority schedulers. Other works try to eliminate or reduce the inter-task interference by using hardware and software techniques [7–10], but they do not face the intrinsic interference problem. Additionally, in some cases, the extrinsic interference is only reduced, and therefore, the predictability problem of the cache-related preemption delay remains unresolved.

This work presents an integrated approach, based on a previous work [11], that offers *full predictability* for WCET estimation, and a bounded value of extrinsic interference under the Earliest Deadline First (EDF) scheduler.

First goal is achieved using instructions to manage cache, like selective preload (cache fill) and cache locking. These instructions are present on currently available processors. The way these characteristics are used offers the possibility to use a simple schedulability analysis joined with accurate estimations of cache performance.

The method here presented is based in the ability of several processors to disable or lock the cache, precluding the replacement of its contents but allowing references to the data or instruction already stored in cache. In this scenario, execution time of instructions is constant for each instance, and preemptions do not modify the cache contents. This way, intra-task and inter-task interference are eliminated since cache content remains unchanged during all system operation, and only a temporal cache buffer have to be taken into account in the schedulability analysis. Such a temporal buffer is introduced to improve temporal behaviour of the instructions not preloaded into the cache.

The rest of the paper is organised as follows: next section shows the hardware necessary to reach both predictability and the best possible performance. Section 3 is devoted to schedulability analysis, and the algorithms used to calculate the WCET and the schedulability analysis are presented when a locking cache is used. In section 4, the genetic algorithm to select the best set of instructions to load in cache is presented. Then, the experimental results are explained. And finally, conclusions and future work are described.

2 System Overview

Several processor offers the ability to lock cache memory contents, like Intel-960, some x86 family processors, Motorola MPC7400, Integrated Device Technology 79R4650 and 79RC64574, and others. Each of these processors implements cache locking in a different ways, allowing to lock the entire cache, only a part, or in a per-line basis. But in all cases, a portion of cache locked will be not selected later for refill by other data or instruction, remaining its contents unchanged.

The IDT-79R4650 cache schema offers an 8KB, two-set associative instruction cache. Also, the processor offers the instruction 'cache fill' instruction to selective load cache contents. However, this processor allows locking only one set of cache, leaving unlocked the other cache set. Since the main objective of this work is to reach a deterministic cache, locking the entire cache is needed. In the MPC7400 is possible to lock the entire cache, using a one-cache-line size buffer to temporally store instructions not loaded in cache, improving sequential access to these addresses. The problem with this processor is that not selective load of cache contents is available. This way, in this work, a merge of the two above processor is proposed, resulting in a cache system with the following characteristics:

- Cache can be totally locked or unlocked. When cache is locked, there are no new tag allocations.
- If the processor addresses an instruction that is in the locked cache, this instruction is served from cache.
- If the processor addresses an instruction that is in the temporal buffer, this instruction is served from this buffer in like-cache time.
- If the processor addresses an instruction that is not in the locked cache or temporal buffer, this instruction is served from main memory. Simultaneously, the temporal buffer is filled with that block regarding the address demanded by the processor.
- Cache can be loaded using a cache-fill instruction, selecting the memory block to load it.
- Cache can be locked using cache management instructions.
- Cache may be direct mapped or set associative. Increasing the associative-level may increase the performance of locking caches, but direct-mapped is enough to reach predictability.

Totally locking the cache allows obtaining the maximum possible performance, simultaneously making deterministic the cache. The temporal buffer reduce access time to memory blocks not loaded in cache, since only references to the first instruction in the block produce cache miss.

During system design step, a set of main memory blocks is selected to be loaded and locked in cache. When system start-up, a small routine will load selected blocks in cache, executing cache fill instructions. After last load, the cache is locked. In this way, when tasks begin full operation, the state of cache is known and remains unchanged during all system operation.

3 Schedulability Analysis

The main goal addressed in this paper is predictability. The designer of a real-time system have to be able to predict the timeliness execution of the critical workload before starting the system. This work can be accomplished using an schedulability test at design time.

In dynamic systems, the schedulability test can be performed by checking the system schedulability throughout a short interval named the Initial Critical Interval (ICI) [12]. In this section, this ICI schedulability test is presented and adapted to take into account the extrinsic interference in a dynamic scheduler, like Earliest Deadline First. As the entire instruccion cache is locked, the extrinsic interference is reduced to the refilling of the temporal buffer.

In a real-time system, the critical workload is typically composed by a set of periodic tasks $\mathcal{T}$. This task set is defined by $\mathcal{T} = \{T_i(C_i, D_i, P_i) : i = 1 \dots n\}$ with $1 \leq C_i \leq D_i \leq P_i$, where C_i, D_i and P_i are the worst-case execution time (WCET), relative deadline and period of task T_i, respectively.

The ICI schedulability test is based on two analytical functions $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$:

- **Function** $G_{\mathcal{T}}(t)$: Given a task set $\mathcal{T}$, function $G_{\mathcal{T}}(t)$ accumulates the amount of computing time requested by all activations of tasks in $\mathcal{T}$ from time zero until time t. Formally:

$$G_{\mathcal{T}}(t) = \sum_{i=1}^{n} C_i \left\lceil \frac{t}{P_i} \right\rceil . \tag{1}$$

- **Function** $H_{\mathcal{T}}(t)$: Given a task set $\mathcal{T}$, function $H_{\mathcal{T}}(t)$ is the amount of computing time requested by all activations of tasks in $\mathcal{T}$ whose deadline is less than or equal to t. Formally:

$$H_{\mathcal{T}}(t) = \sum_{i=1}^{n} C_i \left\lfloor \frac{t + P_i - D_i}{P_i} \right\rfloor . \tag{2}$$

  In other words, $H_{\mathcal{T}}(t)$ represents the amount of computing time that the scheduler should have served until time t in order to meet all deadlines.

Using these functions, the initial critical interval, $\mathcal{R}$, can be calculated by using the recursive expression $R_{i+1} = G_{\mathcal{T}}(R_i)$ until $R_i = R_{i+1}$, where $R_0 = 0$. The last value of R_i indicates the ICI $\mathcal{R}$, that represents the first instant when all requests have already been served and no additional requests have been arrived yet.

Once $\mathcal{R}$ has been established, the system schedulability can be ensured if and only if the next expression is true:

$$H_{\mathcal{T}}(t) < t : \forall t, 1 \leq t \leq \mathcal{R} \ .$$

3.1 Extrinsic Interference

The schedulability test presented above does not consider any cache-related preemption delays. Though critical tasks have a portion of their code locked at instruction cache, every time a preemption is performed by the scheduler, the temporal buffer can be filled by the new task code. When the preempted task resumes its execution, it could undergo a penalty due to the possible refilling of the temporal buffer. Since the preemption point is not known a priori, the worst case scenario must be considered. In this case, a task can be preempted while executing a block of instructions from the temporal buffer. So, using the proposed structure of locking cache, the penalty suffered by the preempted task is T_{miss}, where T_{miss} is the time to transfer a block from main memory to the temporal buffer.

To determine the maximum number of preemptions a task can suffer in a dynamic system, and therefore, to calculate the WCET and the response time of a task taking into account these preemptions, is a very difficult problem. However, it is quite easier to determine the number of preemptions a task originates under a given scheduler. This information can be used in the schedulability test to incorporate the cache-related preemption delay into the task responsible for the preemption, instead of incorporating this delay in the task is preempted.

Earliest Deadline First scheduler is privileged scheduler among schedulers based on dynamic priorities: it generates a very low number of preemptions, and these preeemtions can only occur on task arrivals. Therefore, under EDF, a task generates a preemption when it arrives or does not generate any preemption at all. Taking this feature into account, the schedulability functions (1) and (2) remains as follows:

$$G_{\mathcal{T}}(t) = \sum_{i=1}^{n} (C_i + T_{miss}) \left\lceil \frac{t}{P_i} \right\rceil , \tag{3}$$

$$H_{\mathcal{T}}(t) = \sum_{i=1}^{n} (C_i + T_{miss}) \left\lfloor \frac{t + P_i - D_i}{P_i} \right\rfloor . \tag{4}$$

where C_i is the WCET of the task T_i considering the existence of cache and taking into account the blocks this task has locked in cache. Next subsection presents how this can be calculated.

Though the rest of the schedulability test remains unchanged, a very slight optimitation can be performed. It can be taken into account that the task with the largest relative deadline *never* can preempt any task when it activates, because it always has the slowest priority on arrival.

3.2 Worst Case Execution Time

The schedulability test needs the Worst Case Execution Time of each task T_i to accomplish the analysis. This WCET must be calculated considering the existence of cache. In conventional caches this is a hard problem, because two

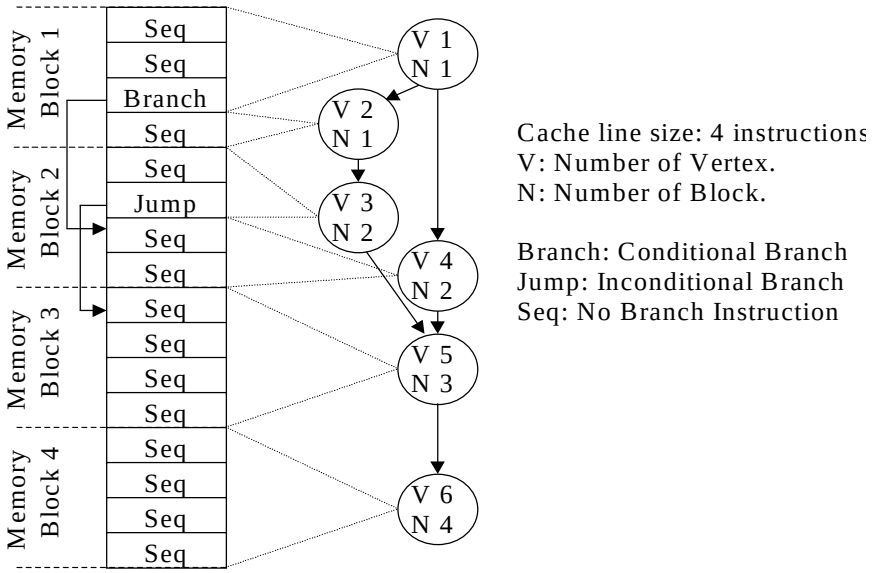

Fig. 1. Example of c-cfg

execution of the same instruction must take different temporal cost. But in here presented cache scheme, an instruction will be in cache always, or never will be into, thus its execution time is always constant. To calculate the WCET of a task, the timing analysis presented in [13] is modified to taking account the presence of the locking cache. This analysis is based on the concept of Control Flow Graph of a task.

This work presents an extended Control Flow Graph, called Cached-Control Flow Graph (c-cfg), that takes into account cache line boundaries. In this c-cfg, a vertex is a sequence of instructions without flow break, and all instructions on a vertex map in the same cache line. This model differs from conventional CFG in the meaning of vertex, since the c-cfg models not only the flow control of the task but also how the task is affected from the point of view of cache structure. Figure 1 illustrates an example.

This c-cfg can be represented with a simple expression that can be evaluated to obtain the task WCET. Figure 2 shows the expression for three basic c-cfg, and Figure 3 shows an example. In these expressions, E_i represents the execution time of vertex V_i.

Task's WCET can be calculated evaluating the expression, considering the execution time of each vertex. The execution time of a vertex depends on the number of instructions into the vertex and the cache state when the vertex is executed. In a locked cache, the cache state remain unchanged, so the execution time of a vertex is constant for all executions: the vertex is always loaded into

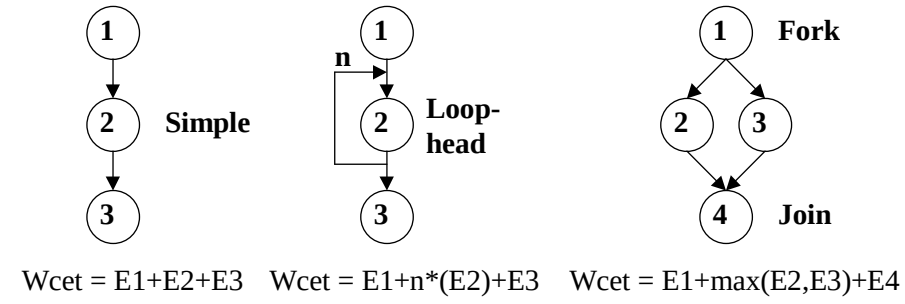

Fig. 2. Expressions for three basic structures

the cache or it will never be. So, the execution time of a vertex can be calculate as follows:

- For a vertex V_i loaded and locked in cache, its execution time E_i is: $E_i = T_{hit} \cdot I_i$.
- For a vertex V_i not loaded nor locked in cache, its execution time E_i is: $E_i = T_{miss} + (T_{hit} \cdot I_i)$

where I_i is the number of instructions of vertex V_i, T_{hit} is the execution time of an instruction that is in cache, and T_{miss} is the time to transfer a block from main memory to the temporal buffer.

The execution time of vertexes can be directly used in the c-cfg expression to obtain the WCET of the task, giving an upper bound value, since execution time is now non cache-dependent. The existence of a temporal buffer may introduce, in some cases, a light error in the WCET estimation.

4 Selecting Blocks to Load and Lock in Cache

Performance improvements due to use of cache memories are very significant, and real-time systems should take advantage of it. Randomly loading and locking instructions in cache offers predictability but not guarantee good response time of the tasks. In order to reach both goals, a predictable cache and a cache performance close to the usual one, instructions to be loaded must be carefully selected, trying to find the best possible scenario. This scenario is a set of main memory blocks locked in cache that provides the minimum possible execution time, thus providing the minimum possible response time for a set of tasks.

Although there are several possibilities to select instructions to be locked, it is not easy to isolate an instruction and evaluate the impact of locking it in cache over the system behaviour, due to interacts between tasks. Response time of task is mainly related to the task's structure, but also how tasks are scheduled in the system concerns to the response time. Exhaustive search, including branch and bound, presents an intractable computational cost, since the number of possible

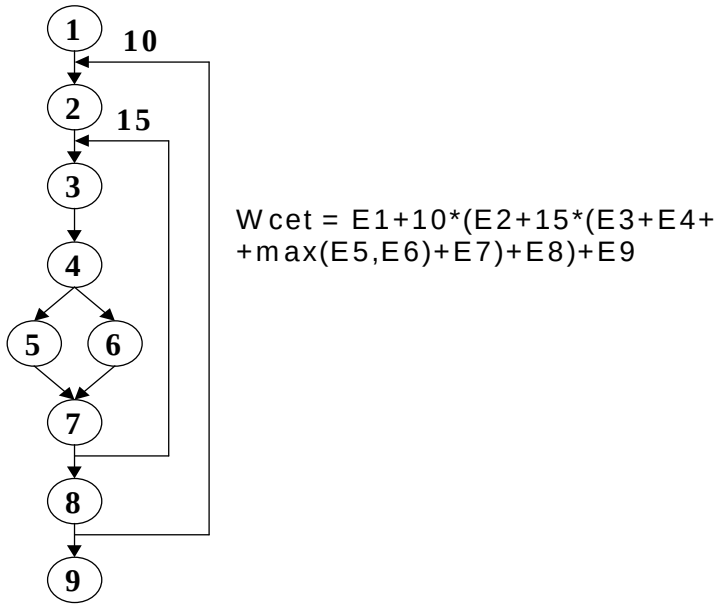

Fig. 3. Example of expression

solutions is very large. Genetic algorithms [14], performing a randomly-directed search, can be used in this problem, finding a sub-optimal solution within an acceptable computational time. The genetic algorithm used in this work is the evolution of a previous version presented in [11]. The main characteristics of the new algorithm are described next.

Each block of a task can be locked or not in cache. An individual represents the state of all blocks of all tasks in the system in one chromosome, where a chromosome is a set of genes. Each gene has a size of only one bit and represents the block state. The population is a set of individuals.

Fitness function must guide the genetic algorithm evolution, helping to find the best solution. The fitness function must have three main characteristic: low computational cost, find the best solution, and find this solution in fewer iterations. It is hard to find a fitness function that agree these characteristics, and usually it is a complex function. In this work, the used fitness function is the result of applying the schedulability test described in previous section to each individual, considering the state – locked or not – of the blocks. WCET for schedulability test is estimated using the WCET expressions described in previous section. From the fitness function four types of results are obtained:

– Schedulable system, with number of locked blocks minor or equal to the cache size. This is a valid individual.

– Schedulable system, with number of locked blocks greater than cache size. This is a non-valid individual.
– No schedulable system, with number of locked blocks minor or equal to the cache size. This is a very bad solution, but a valid individual.
– No schedulable system, with number of locked blocks greater than cache size. This is a non-valid individual.

Also, fitness function returns for schedulable individuals the system utilisation, and for not schedulable individuals it returns a factor indicating how bad is the individual (distance between failure time and the ICI). The existence of invalid and non-schedulable individuals precludes the use of direct probability setting as function of fitness value. This way, individuals are arranged in three segments: higher positions for schedulable-and-valid individuals, following valid-non-schedulable individuals, and lower positions for invalid individuals. Into first segment, schedulable-and-valid individuals are arranged as function of its utilisation (lower utilisation, higher position). Into second segment, valid-non-schedulable individuals are arranged as a function of its factor of failure (higher factor, higher position). Finally, invalid individuals are arranged as function of its number of locked blocks (lower number of blocks, higher position). Once all individuals are well arranged, selection probability for crossover is set as function of position. This allows including, with low probability, both non-schedulable and non-valid individuals that help to increase the variability of the algorithm.

Crossover is performed choosing randomly a gene that divides the individual into two parts, and exchanging the parts of two individuals, making two new individuals. This process is repeated until the number of new individuals make equal the population size.

Mutation is applied in a gene-basis to these new individuals in three ways:

– For individuals with number of locked blocks greater than cache size, mutation randomly eliminates blocks from the set of locked-blocks.
– For individuals with number of locked blocks smaller than cache size, mutation randomly adds blocks to the set of locked-blocks.
– For individuals with number of locked blocks equal than cache size, mutation randomly exchange blocks, leaving unchanged the number of locked blocks.

In order to guarantee the use of a direct-mapped locking cache, after the previous mutation, the algorithm looks if the set of locked blocks do not fit in a direct-mapped cache, randomly exchanging locked blocks, when needed, making them fit in a direct-mapped cache.

A new population is building with the individuals obtained from mutation, and process is repeated a prior-defined number of times. For the accomplished experiments presented further in this paper, the number of iterations is established in 2.000, with a population of 200 individuals.

The genetic algorithm solves, at the same time, the problem of selecting main memory blocks to load and lock in cache, and also, the schedulability analysis, since the result from the fitness function for a valid individual is the response of schedulability test.

5 Experimental Results

Above presented analysis allows to bound execution time interferences due to cache related issues. However, although the effects of using the proposed cache scheme can be bounded and incorparated to the schedulability analysis, the performance advantages obtained from using cache memories in a predictable way should be analysed.

Experimental results presented in this section show that preload and locking instructions in cache not only makes the system predictable: it also offers a performance close to the traditional caches (direct-mapped or set-associative) with LRU or Pseudo LRU replacement algorithm.

To make experiments, the SPIM tool [15], a MIPS R2000 simulator is used. The SPIM does not include neither cache nor multitask, so modifications to include an instruction cache, multitasking (simulated and controlled by the simulator and not by the O.S.) and to obtain execution times has been made to the original version of SPIM. Since this simulator does not include any architectural improvement, cache effects can be analysed without interference. The routine to load and lock in cache the selected instructions is incorporated in the simulator. Tasks used in experiments are artificially created to stress the proposed cache scheme. Main parameters of task are defined, like number of loops and nesting level, size of tasks, size of loops, number of if-then-else structures and its respective sizes. These parameters are fixed or randomly selected. A simple tool is used to create tasks. The workload of any task may be a single loop, if-then-else structures, nested loops, streamline code, or any mix of these. The size of task code range from near 64 Kb to around 1Kb.

Each experiment is composed of a set of tasks and a cache size, ranging from three to eight tasks and cache sizes from 1 Kbyte to 64 Kbytes. This way, the two extreme scenarios are presented: code size much greater than cache size (64:1) and code size lower than cache size. Each experiment is simulated using direct-mapped, two-set associative, four-set associative and full associative cache, calculating the system utilisation U_{cache}. For all cases, line size is 16 bytes (four instructions). Time to transfer a block from main memory to the temporal buffer is 10 cycles ($T_{miss} = 10$). Execution of any instruction from the cache is 1 cycle, and execution of any instruction from the temporal buffer is also 1 cycle. For each experiment, the system utilisation is estimated using the genetic algorithm $U_{estimated}$, and simulated in a locking cache using the blocks selected by the genetic algorithm $U_{locking}$.

Figure 4 presents the overestimation in the estimated utilisation by the genetic algorithm, respect the actual utilisation (simulated) of the system when locking cache is used. (($U_{estimated}/U_{locking}$) -1). Each bar represents the number of experiments with percentage of overestimation that lies in the interval of the x-axis (i.e., 36 experiments have an overestimation between 0,01% and 0,05%). This figure shows that the estimated utilisation is quite accurate: The overestimation is always below the 0,5%. So, pessimism introduced in WCET calculation and schedulability analysis is not significant.

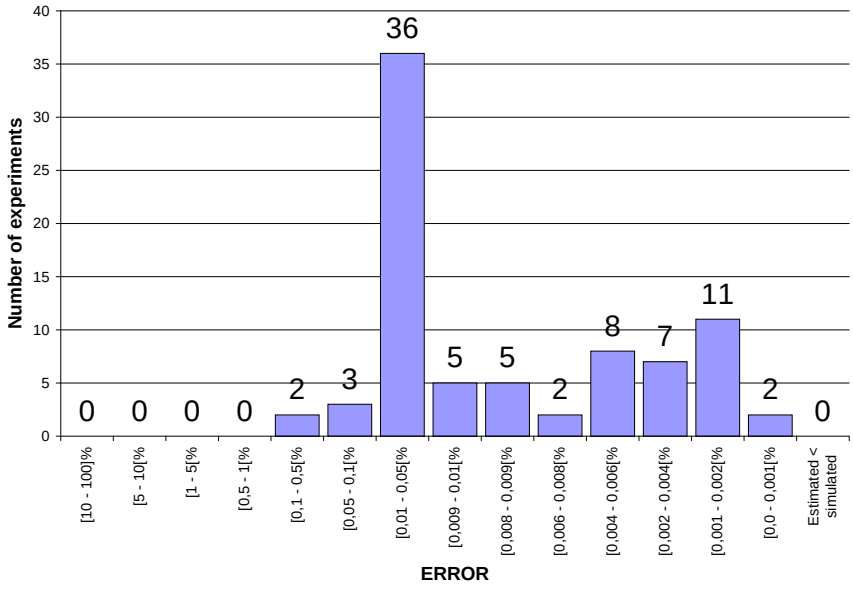

Fig. 4. Overestimation estimated by the genetic algorithm

Figure 5 shows the accumulated frequency. Accumulated number of experiments for the given overestimation between simulated and estimated system utilisation using locking cache. Axis-y value is the percentage of experiments with an overestimation lower than axis-x value. It can be observed that more than 90% of the experiments present an overestimation below 0,05%.

Regarding the performance of the locking cache, Figure 6 compares the system utilisation with or without locking cache. Conventional cache uses the mapping function that obtains the best performance for each case.

The figure depicts the performance ratio: simulation of actual system utilisation with the best conventional-cache arrangement, versus the estimated system utilisation obtained by the genetic algorithm with a locking cache ($U_{cache}/U_{estimated}$). Tasks are grouped regarding this ratio. Each bar represents the number of experiments with performance ratio ($U_{cache}/U_{estimated}$) that lies in the interval of the x-axis.

Figure 7 draws accumulative values of previous figure. Axis-y value is the percentage of experiments with performance ratio greater than axis-x value. For around 50% of the experiments, the system utilisation is equal or lower using locking cache, and in more than 60% of cases the performance loss is negligible. In these cases, the worst case response time (WCRT) is not only bounded, furthermore it makes the WCRT lower than execution time in a system with a normal cache.

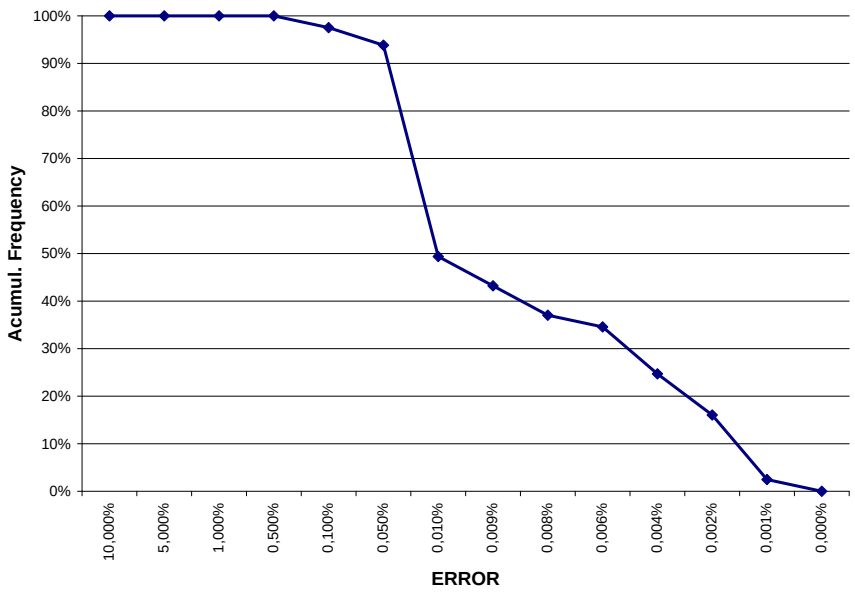

Fig. 5. Accumulated frequency of overestimation in estimated utilisation

From the obtained results, we can conclude that the proposed cache scheme is predictable, and it allows the application of EDF schedulability analysis in systems with cache. The estimated utilisation is an upperbound of the actual utilisation using locking cache: ($U_{locking} < U_{estimated}$ for all experiments). With this technique, the predictability is obtained in many cases without performance loss ($U_{estimated} <= U_{cache}$) for around 60% of experiments).

6 Conclusions

This work presents a novel technique that uses locking caches in the context of real-time systems with EDF schedulers. In addition, algorithms to analyse the proposed system are described. Compared to known techniques to achieve cache predictability in Real-Time systems, this solution completely eliminates the intrinsic cache interference, and gives a bounded value of the extrinsic one.

This technique allows real-time systems with dynamic scheduling profit from the great performance increase produced by cache memories. And this is accomplished in a practical way, since the designer can easily analyse the system to accomplish the schedulability test. In addition, the architecture is compatible with other techniques to improve performance, like segmentation, precluding the consideration of the complex interrelations amongst these techniques and the cache.

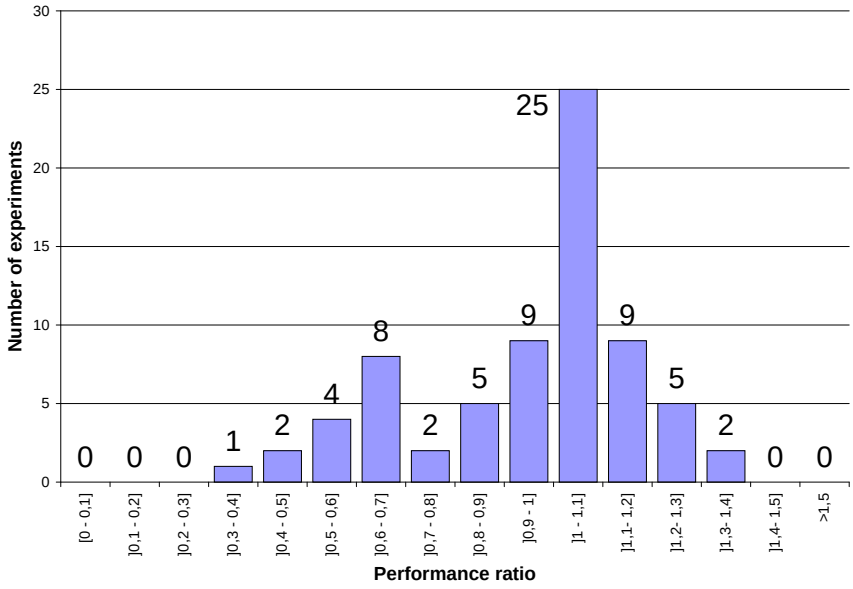

Fig. 6. Performance ratio obtained when using locking cache.

This approach is very effective from the performance point of view. Simulations results show that for around 60% of experiments the performance achieved by using locking caches is almost similar to the one obtained with conventional caches (without taking care of determinism).

The hardware resources required to implement this scheme are available in some contemporary processors. To obtain the best results, some minor changes have been proposed. These changes do not present difficulties in terms of technical complexity and production.

This work has also presented an algorithm to select the contents of the cache. This selection delivers the best performance. The algorithm also calculates the WCET and performs the schedulalibility analysis.

References

1. Healy, C.A., Arnold, R.D., Mueller, F., Whalley, D., Harmon, M.G.: Bounding pipeline and instruction cache performance. IEEE Transaction on Computers **48** (1999) 53–70
2. Lim, S.S., Bae, Y.H., Jang, G.T., Rhee, B.D., Min, S.L., Park, C.Y., Shin, H., Park, K., Kim, C.S.: An accurate worst case timing analysis technique for risc processors. In: Proc. of the 15th IEEE Real- Time Systems Symposium. (1994)

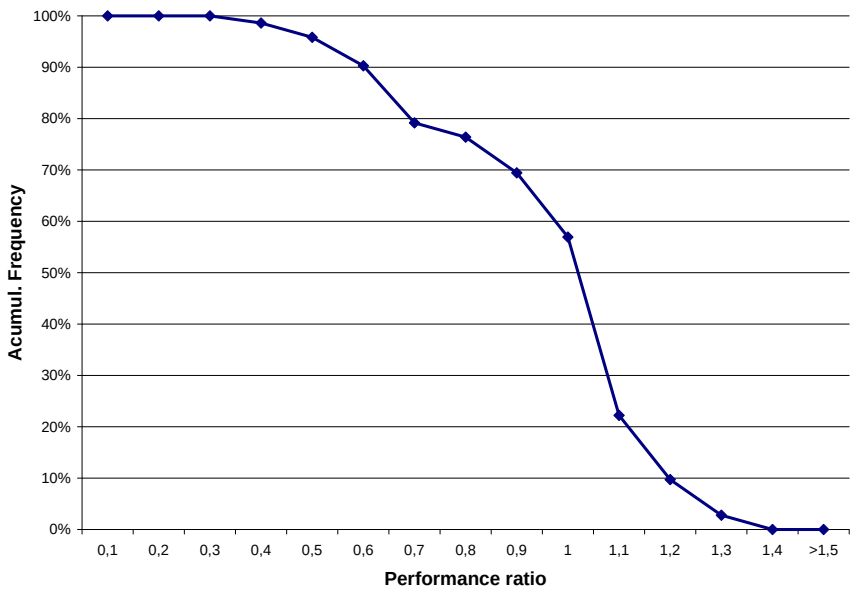

Fig. 7. Accumulative performance ratio when using locking cache.

3. Li, Y.S., Malik, S., Wolfe., A.: Cache modeling for real-time software: Beyond direct mapped instruction caches. In: Proc. of the 17th IEEE Real-Time Systems Symposium. (1996)
4. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache behaviour prediction by abstract interpretation. In: Proc. of Static Analysis Symposium 1996. Number 1145, Lecture Notes in Computer Science (1996) 52–66
5. Busquets, J.V., Serrano, J.J., Ors, R., Gil, P., Wellings, A.: Adding instruction cache effect to an exact schedulability analysis of preemptive real- time systems. In: Proceedings of the IEEE Euromicro Workshop on Real-Time Systems. (1996)
6. Lee, C.G., Hahn, J., Seo, Y.M., Min, S.L., Ha, R., Hong, S., Park, C.Y., Lee, M., Kim, C.S.: Enhaced analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In: Proceedings of the 18th IEEE Real-Time System Symposium. (1997)
7. Kirk, D.B.: Smart (strategic memory allocation for real-time) cache design. In: Proceedings of the 10th IEEE Real-Time Systems Symposium. (1989)
8. Liedtke, J., Härtig, H., Hohmuth, M.: Os- controlled cache predictability for real-time systems. In: Proceedings of the IEEE Real-Time Technology and Applications Symposium. (1997)
9. Busquets, J.V., Serrano, J.J., Wellings, A.: Hybrid instruction cache partitioning for preemptive real-time systems. In: IEEE Euromicro Workshop on real-time Systems. (1997)
10. Wolfe, A.: Software-based cache partitioning for real-time applications. In: Proceedings of the 3th International Workshop on Responsive Computer Systems. (1993)

11. Martí, A., Pérez, A., Perles, A., Busquets, J.: Using genetics algorithms in content selection for locking- caches. In: IAESTED International Conference on Applied Informatics, Acta Press (2001)
12. Ripoll, I., Crespo, A., Mok, A.: Improvements in feasibility testing for real-time tasks. The Journal of Real-Time Systems **11** (1996) 19–39
13. Shaw, A.: Reasoning about time in higher- level language software. IEEE Transaction on Software Engineering **15** (1989)
14. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and machine Learning. Addison- Wesley Co. (1989)
15. Patterson, D., Hennessy, J.L.: Computer Organization and Design. The Hardware/Software Interface. Morgan Kaufmann, San Mateo (1994)