

# Static Use of Locking Caches in Multitask Preemptive Real-Time Systems

Marti Campoy, A. Perles Ivars, J. V. Busquets Mataix  
Computer Engineering Department. Technical University of Valencia, Spain

**Abstract--In multitask, preemptive real-time systems, the use of cache memories make difficult the estimation of the response time of tasks, due to the dynamic, adaptive and non-predictable behaviour of cache memories. But many embedded and critical applications need the increase of performance provided by cache memories.**

**This work presents a comprehensive method to attain predictability on the use of caches in real-time systems. Locking cache mechanisms allow to load and lock the content of the cache to ensure it will remain unchanged during execution. By ensuring this, the cache is totally predictable and it allows a more accurate estimation of response time of tasks. In addition, conventional algorithms can be used to accomplish the schedulability analysis.**

**Nowadays, locking cache scheme is present in several commercial processors, and only minor hardware modifications would be necessary in order to obtain the best performance. To select the contents to be preloaded in cache, a genetic algorithm has been developed. This algorithm selects the set of instructions to be locked in cache that give the better performance. It estimates a tight upperbound of the response time of tasks, making simultaneously the schedulability analysis.**

**Experimental results indicate that this scheme is fully predictable, and the performance loss is negligible for around 60% of the tasks. In some cases, the estimated worst case performance using locking cache is better than using a standard cache.**

## 1 INTRODUCTION

Modern microprocessors include cache memories in its memory hierarchy to increase system performance. General-purpose systems directly benefit from this architectural improvement, but specific systems, like hard real-time systems, need special hardware resources and/or system analysis to guarantee the time correctness of system behaviour when cache memories are present.

In multitask, preemptive real-time systems, response time of any task must be calculated a priori, in order to guarantee that all the tasks meets their deadlines. Using cache memories in such systems presents two problems, due to the dynamic, adaptive and non-predictable behaviour of cache memories.

The first problem is to calculate the Worst Case Execution Time (WCET) due to the intra-task or intrinsic interference. Intra-task interference arises when a task removes its own instructions from cache due to conflict and capacity misses. When removed instructions are executed again, a cache miss increases the execution time of the task. This way, the delay caused by cache memory interference must be included in the WCET calculation.

The second problem is to calculate the task response time

due to the inter-task or extrinsic interference. Inter-task interference arises in preemptive multitasking systems, when a task displaces from the cache the working set of any other task of the system. When the preempted task resumes execution, a burst of cache misses increase its response time. This effect, called cache-refill penalty or cache-related preemption delay must be considered in the schedulability analysis, because increases the task execution time over the precalculated WCET.

Several solutions have been proposed to use cache memories in real-time systems. [1,2,3,4] analyse the cache behaviour to estimate the task execution time considering the intra-task interference. [5,6,7] analyse the cache behaviour to estimate the task response time considering the inter-task interference, using a precalculated cached WCET. [8,9,10,11] use hardware and software techniques to eliminate or reduce the inter-task interference, but they do not face the intrinsic interference problem.

The main drawback of previous solutions is that they only solve one side of the problem: intrinsic interference or extrinsic interference, whilst assuming the other face is currently solved. Added to this, in order to get accurate estimations of response time, complex analysis techniques are needed.

The main goal of this work is to present a cache scheme that offers full predictability for both levels, WCET estimation and schedulability analysis, using instructions to manage cache, present in modern processors, like selective preload (cache fill) and cache locking (in some cases called cache isolated). In this paper, a method to use these characteristics is presented, dealing with three major goals:

- Both intrinsic and extrinsic interference are eliminated, making cache behaviour totally predictable, allowing a simple analysis, even when other architecture improvements are used, since memory access delays are constant.
- Necessary hardware is nowadays present in several commercial processors, and only minor hardware modifications are mandatory in order to get the best performance.
- In several cases, the preload-and-lock method presents the same or better performance than conventional use of cache.

The method here presented is based in the ability of several processors to disable or lock the cache, precluding the removal of its contents but allowing references to the data or instruction already stored in cache. Locking the cache eliminates the interference, both intrinsic and extrinsic, since cache content remains unchanged during

system operation. This functionality joined with instructions for selective preloading the cache, makes the contents of cache well-known before execution, eliminating the cache unpredictability.

This work only considers instruction cache, without regard to other architecture improvements.

The paper is organised as follows: next point shows the hardware necessary to reach both predictability and the better possible performance. In Schedulability analysis the algorithms to calculate the WCET and the Response time are presented when a locking cache is used. Following, the genetic algorithm to select the best set of instructions to load in cache is described. Finally, experimental results are presented.

## 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 processor implements cache locking in several 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 not be selected later for refill by other data or instruction, remaining its contents unchanged.

Our work is based in the IDT-79R4650 cache schema. This processor offers an 8KB, two-set associative instruction cache. Also, the processor offers the instruction "cache fill" to selective load cache contents. However, this processor allows locking only one set of cache, leaving unlocked the other. Since the main objective of this work is to reach a deterministic cache, we need to lock the entire cache. In the MPC7400 it 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. However, the selective load of cache contents is not 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 isn't in the locked cache or temporal buffer, this instruction is served from main memory. Temporally buffer is filled with the 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 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

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

During system start-up, a small routine is executed to preload the cache. This routine is basically a loop that read from a table the addresses of instructions (in reality addresses of main memory blocks) and preloads it executing cache fill instruction. To eliminate interference between instructions to be preloaded and the routine, this may be allocated in a non-cacheable area of main memory. After preload the last block, cache is locked. Preloaded instructions can belong to any task of the system, and may be large consecutive instruction sequences or small, individual separate blocks. When the system begins its full-operational execution, the instruction cache is loaded with a well-know set of instructions, and its contents will never change, eliminating both intra-task and inter-task interference.

## 3 SCHEDULABILITY ANALYSIS

In order to guarantee that a multitask, preemptive system is schedulable, it may be used the Response Time Analysis (RTA). Equation 1 shows the expression of RTA, used to calculate, in several iterations, the response time of each task in the system. The schedulability analysis is performed trivially comparing this value with the task deadline. In this equation,  $w_i$  denotes the response time of task  $\tau_i$ ,  $C_i$  is the WCET of  $\tau_i$  without preemptions,  $B_i$  denotes the time task  $\tau_i$  is blocked and  $T_j$  is the period of task  $\tau_j$  and  $hp(i)$  is the set of tasks with higher priority than task  $\tau_i$ .

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil x C_j \quad (1)$$

To take into account the extrinsic interference of the cache, we use an analysis derived from RTA, called CRTA [5]. Equation 2 shows the expression of CRTA, where  $C_i$  is the WCET of  $\tau_i$  without preemptions but considering cache effect, and  $\gamma_j$  is the rise in the response time that task  $\tau_i$  experience due to task  $\tau_j$ . But in the here presented system, inter-task interference doesn't exist since cache contents remain unchanged during task switch, except for a light extrinsic-interference introduced by the temporal buffer. Since the preemption point is not known a priori, the CRTA must consider the worst scenario. In this case, a task can be preempted while executing a block of instructions from temporal buffer. The contents of this buffer will be removed and the preempted task must refill the temporal buffer when resume execution after preemption, increasing its execution time. Using locking cache, the value of  $\gamma_j$  is  $T_{miss}$ , where  $T_{miss}$  is the time to transfer a block from main memory to the temporal buffer.

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_j^n}{T_j} \right\rceil x(C_j + \gamma_j) \quad (2)$$

In the expression of CRTA,  $C_i$  represents the WCET of task  $\tau_i$ , and this must be calculated considering the existence of cache. To calculate the WCET of a task taking into account the presence of locking cache, a modified timing analysis [12] is proposed.

From the task's Control Flow Graph and machine code, an extended Control Flow Graph, called Cached-Control Flow Graph (c-cfg), is created. 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 a vertex, since the c-cfg models not only the task's paths but also how the cache is used. Figure 1 illustrates an example.

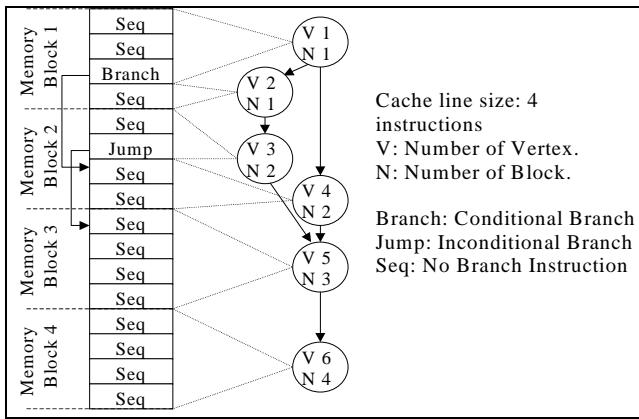


Figure 1. Example of c-cfg.

This c-cfg can be represented with a simple string, an 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  $V_i$ .

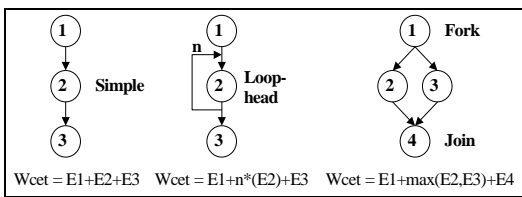


Figure 2. Expressions for three basic structures.

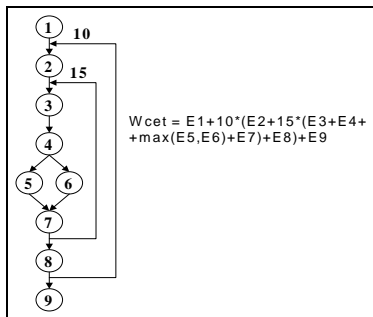


Figure 3. Example of expression.

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 cached system, the cache state can change for each execution of a vertex, so the execution time is not constant. But in a locked cache, the cache state remain unchanged, so the execution time of a vertex is constant for all executions. This way, the execution time of a vertex can be calculated as follows:

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

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 vertex execution time 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.

#### 4 SELECTING BLOCKS TO LOAD AND LOCK IN CACHE.

The increase of performance due to the use of cache memories is very significant, and real-time systems must take advantage of it. Randomly loading and locking instructions in cache offers predictability but do not guarantee good response time of tasks. In order to reach both goals, a full-predictable cache and a like-cache performance, instructions to be loaded must be carefully selected, looking for the best 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.

It is not easy to find an algorithm to directly select blocks to load and lock in cache. Instructions locked in cache will execute faster. Thus, instructions inside loops must be selected. But in preemptive, multitask systems, execution time of tasks depends on the execution time of higher priority tasks. This way, the number of iterations an instruction is executed is not enough to decide either to lock or not the instruction. The effect in other task must be considered when selecting an instruction to lock it in cache. Another problem selecting instructions to be locked is the existence of alternate paths. In the worst execution case, some instructions in alternate paths will never be executed. Thus locking instructions from these paths do not reduce the task's WCET nor the response time of system's tasks. The algorithm must select these blocks that reduce not only the WCET but also the response time of all tasks. Although there are several pointers 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's response time. Exhaustive search, including branch and bound, presents an intractable computational cost, since the number of possible solutions is very large. As the problem is not

monotonic, algorithms like hill climbing are not useful.

Genetic algorithms [13], performing a randomly-directed search, can be used in this problem, finding a sub-optimal solution within an acceptable computational time. The algorithm provides the set of blocks (since the block is the minimum unit of information that can be transferred from main memory to cache, the algorithm must work with blocks and not with individual instructions), an estimation of the WCET of each task executing in a locked cache with the set of blocks loaded and locked, and the response time of all tasks considering the WCET estimated using the locking cache. The main characteristics of the developed algorithm are described:

Each block 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 of one bit size represents the block state. The population is a set of individuals. The fitness function is the weighted average of the response time of tasks considering the state—either locked or not—of the blocks. The response time of tasks is calculated using the CRTA and the WCET expressions described in previous section. From the fitness function we obtain four types of results:

- Finite average value, with number of locked blocks minor or equal to the cache size. This is a valid individual (solution).
- Finite average value, with number of locked blocks greater than cache size. This is a non-valid individual.
- Infinite average value, with number of locked blocks minor or equal to the cache size. Sometimes, some tasks never end its execution before deadline, and this is computed as infinite response time. This is a very bad solution, but a valid individual.
- Infinite average value, with number of locked blocks greater than cache size. This is a non-valid individual.

The existence of invalid individuals precludes the use of direct probability setting as function of fitness value. This way, individuals are arranged regarding its degree of validity. We consider both the number of locked blocks and the average value to arrange both valid and non-valid individuals. Once all individuals are well arranged, selection probability for crossover is set as function of position. This allows including, with low probability, non-valid individuals that help to increase the variability of the algorithm.

Crossover is performed choosing a gene randomly 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 equals the population size. Mutation is applied in a gene-basis to these new individuals in three ways:

- For individual with number of locked blocks greater than cache size, mutation randomly eliminates blocks from the set of locked-blocks.
- For individual with number of locked blocks smaller than cache size, mutation randomly adds blocks to the set of locked-blocks.

- For individual with number of locked blocks equal than cache size, mutation randomly exchange blocks, leaving unchanged the number of locked blocks.

A new population is built 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 5.000, and the response time of the algorithm is lower than 10 minutes in a medium-range personal computer.

A detailed description of the algorithm can be found in [14]. In this work, a new restriction is added to the genetic algorithm: after mutation, the algorithm changes, when necessary, the locked blocks, in order to guarantee the use of a direct-mapped locking cache.

The genetic algorithm, giving the set of memory blocks to preload and lock in cache, and the response time of each task locking these set of instructions, solves, at the same time, the problem of block selection and the schedulability analysis, since the response time values obtained from the genetic algorithm are an upperbound of the task response time, and can be compared with task deadlines to validity the system schedulability.

## 5 EXPERIMENTAL RESULTS

Experimental results must show if preload and locking instructions in cache makes the system predictable and also if the proposed scheme obtain similar performance than 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, thus modifications has been made to the original version of SPIM to include an instruction cache, multitasking (simulated and controlled by the simulator and not by the O.S.) and to obtain execution times. Since this simulator does not include any architectural improvement, cache effects can be analyzed without interference.

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. Period of tasks is hand-defined to make the system schedulable, and deadline is equal to period. Finally, the priority is assigned by Rate Monotonic (the shorter the period the higher the priority). 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 a task code may be large or short.

Each experiment is composed of a set of tasks, ranging from three to eight tasks. Each experiment is simulated using direct-mapped, two-set associative, four set associative and fully associative cache, with cache sizes ranging from 1 Kbyte to 64 Kbytes. For all cases, line size is 16 bytes (four instructions). Execution of any instruction from main memory is 10 cycles, and execution of any

instruction from cache (or temporal buffer) is 1 cycle. For each experiment, the response time of each task is estimated using the genetic algorithm, and simulated in a locking cache using the selected blocks by the genetic algorithm.

Figure 4 presents the error between the response time of every task, either estimated by the genetic algorithm or simulated using a locked cache. Experiments collect more than 800 executions. Each bar represents the number of tasks with estimated error that lies in the interval of the x-axis. This figure shows that:

- The proposed technique is conservative: the estimated response time is always larger than the simulated one.
- The estimated response time is tight. It never surpasses the 5%, and only for 7 tasks, it is above 0.5%.

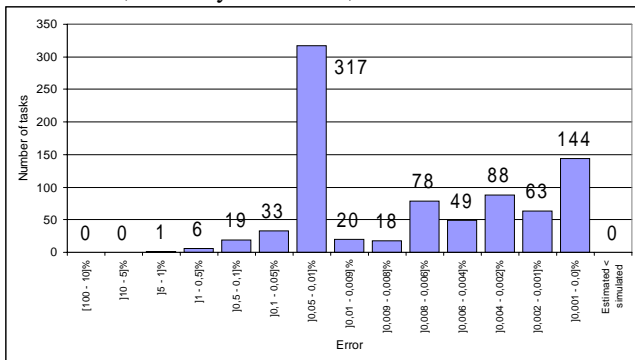


Figure 4. Error between simulated and estimated task response time using locking cache.

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

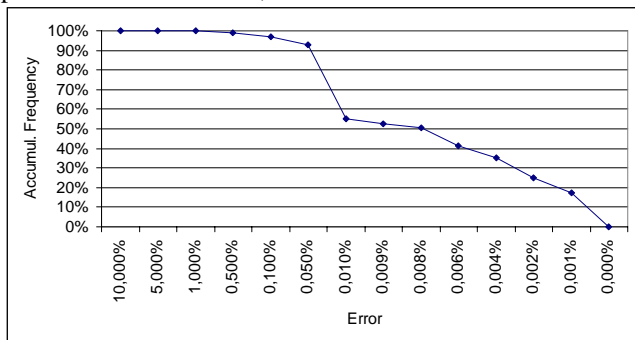


Figure 5. Accumulated frequency of error between estimated and simulated response time.

Regarding the performance of the locking cache, Figure 6 compares the task response time 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 task response time with the best cache arrangement, versus the estimated task response time obtained by the genetic algorithm with a locking cache. Tasks are grouped regarding this ratio.

Figure 7 draws accumulative values of previous figure.

For around 60% of the tasks, the response time is equal or better than using locking cache.

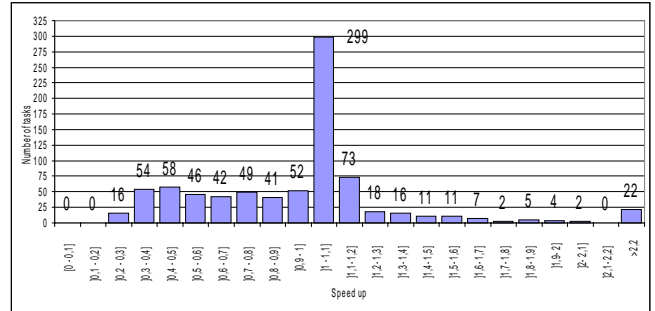


Figure 6. Task speedup obtained when using locking cache. Each bar represents the number of tasks with speedup that lies in the interval of the x-axis.

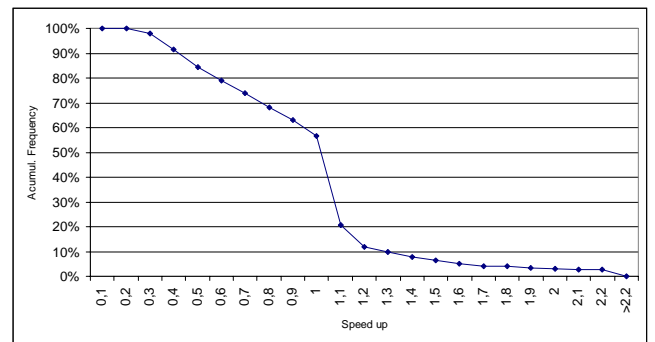


Figure 7. Accumulative task speedup when using locking cache. Axis-y value is the percentage of tasks with speedup greater than axis-x value

Figure 8 and Figure 9 present the speedup considering the average of the response time of each task in each experiment instead of individual tasks. The interest of these figures is to show the global behaviour of the system, because in the same experiment, locking cache gives better performance for some tasks and worse performance for others tasks. For near 60% of experiments, the average of tasks response time using locking cache is equal or better than using any other cache scheme. Only for a 20% of experiments, speedup is lower than 0,6.

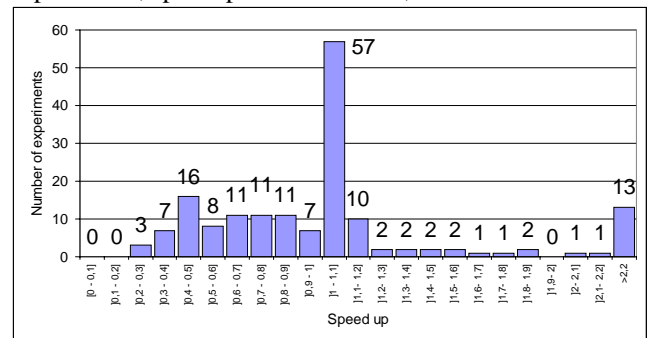


Figure 8. Experiment speedup obtained when using locking cache. Each bar represents the number of experiments with speedup that lies in the interval of the x-axis.

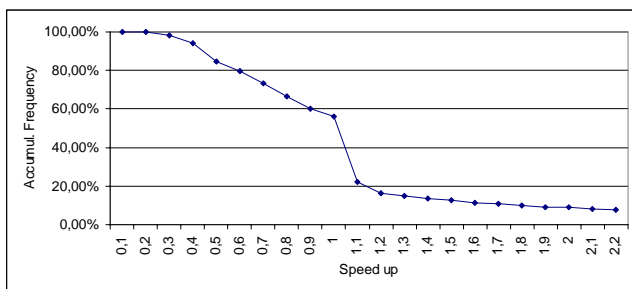


Figure 9. Accumulative experiment speedup when using locking cache. Axis-y value is the percentage of experiments with speedup greater than axis-x value

From the obtained results, we can conclude:

1. The proposed cache scheme is predictable, and it allows the calculation of the response time of tasks in a preemptive scheduler with negligible overestimation
2. With this technique, the predictability is obtained in many cases without performance loss. Moreover, in some cases, the performance is enhanced.

## 6 CONCLUSIONS

This work presents a novel technique that uses locking caches in the context of real-time systems. Compared to known techniques to achieve cache predictability in these systems, our solution completely eliminates the cache interference, either intrinsic or extrinsic.

The benefits of a fully predictable cache are basically two: first, it is practical, since the designer can easily analyse the system to obtain schedulability. Second, the architecture is compatible with other techniques to improve performance, like segmentation, precluding the consideration of the complex interrelations among these techniques and the cache.

This predictability is reached without loss of performance for around 60% of experiments.

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

We have also presented an algorithm to select the contents of the cache. This selection delivers the best performance. The algorithm also calculates the WCET and response time of each task.

## 7 ACKNOWLEDGMENTS

We thank Guillen Bernat, for the automatic code generation. This work has been partly founded by the Spanish Comision Interministerial de Ciencia y Tecnología under project CICYT-TAP 990443-C05-02

## 8 REFERENCES

[1] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley and M. G. Harmon. "Bounding Pipeline and Instruction Cache Performance". IEEE Transaction on Computers. Jan 1999, volume 48, number 1, pages 53-70

- [2] Lim, S. S., Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. Proc. of the 15th IEEE Real-Time Systems Symposium, December 1994.
- [3] Li, Y. S., S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. Proc. of the 17th IEEE Real-Time Systems Symposium, December 1996.
- [4] M. Alt, C. Ferdinand, F. Martin, R. Wilhelm. "Cache Behaviour Prediction by Abstract Interpretation" In Static Analysis Symposium, 1996
- [5] J. V. Busquets, J. J. Serrano, R. Ors, P. Gil, A. Wellings. "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] C. G. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, C. S. Kim. "Enhanced Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling" In Proceedings of the 18th IEEE Real-Time System Symposium, 1997
- [7] A. Martí, X. Molero, A. Perles, F. Rodríguez, J.V. Busquets. "Combined Intrinsic-Extrinsic Cache Analysis for Preemptive Real-Time Systems". Real-Time Programming 2000. 25th IFAC Workshop on Real Time Programming, Pergamon, 2000.
- [8] D. B. Kirk. "SMART (Strategic Memory Allocation for Real-Time) Cache Design" In Proceedings of the 10th IEEE Real-Time Systems Symposium, 1989
- [9] J. Liedtke, H. Härtig, M. Hohmuth. "OS-Controlled Cache Predictability for Real-time Systems" In Proceedings of the IEEE Real-Time Technology and Applications Symposium, 1997
- [10] J. V. Busquets, J. J. Serrano, A. Wellings. "Hybrid Instruction Cache Partitioning for Preemptive Real-Time Systems" In IEEE Euromicro Workshop on real-time Systems, 1997
- [11] A. Wolfe. "Software-Based Cache Partitioning for real-time Applications" Proceedings of the 3th International Workshop on Responsive Computer Systems, 1993
- [12] A. Shaw. "Reasoning About Time in Higher-Level Language Software" IEEE Transaction on Software Engineering, Vol. 15, Num. 7, 1989
- [13] D. E. Goldberg. "Genetic Algorithms in Search, Optimization and machine Learning" Addison-Wesely Co., 1989
- [14] A. Martí, A. Pérez, A. Perles, J.V. Busquets. "Using Genetics Algorithms in Content Selection for Locking-Caches". IAESTED International Conference on Applied Informatics, Acta Press, 2001.
- [15] D. Patterson and J. L. Hennessy. "Computer Organization and Design. The Hardware/Software Interface". Morgan Kaufmann. San Mateo, 1994.