15th Triennial World Congress, Barcelona, Spain

DYNAMIC USE OF LOCKING CACHES IN MULTITASK, PREEMPTIVE REAL-TIME SYSTEMS

A. Marti Campoy, A. Perles Ivars, J. V. Busquets Mataix

Departamento de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia, Spain

Abstract: In multitask, preemptive real-time systems, the use of cache memories makes estimating the response time of tasks difficult, due to the dynamic, adaptive and non-predictable behaviour of cache memories. This work presents a comprehensive method for attaining predictability on the use of caches in real-time systems through the use of locking caches, which ensure cache contents will remain unchanged during the execution of each task. Nowadays, locking caches are present in several commercial processors. In order to select the contents to be locked in cache, a genetic algorithm has been developed. Experimental results indicate that this scheme has a high level of predictability, and that the performance loss is negligible for around 70% of the tasks. *Copyright © 2002 IFAC*

Keywords: Cache Memories, Response Times, Execution Times, Algorithms

1. INTRODUCTION[1]

Modern microprocessors include cache memories in their memory hierarchy to increase system performance. General-purpose systems benefit directly from this architectural improvement, but specific systems, such as hard real-time systems, need additional hardware resources and/or system analysis to guarantee the time correctness of the system's behaviour when cache memories are present. In multitask, preemtive real-time systems, using cache memories presents two problems.

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 the 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 the working set of any other task from the cache. When the preempted task resumes execution, a burst of cache misses increases its execution time. This effect, called cache-refill penalty or cache-related preemption delay must be considered in the schedulability analysis, since it situates task execution time over the precalculated WCET.

Several solutions have been proposed for the use of cache memories in real-time systems. Healy, *et al.* (1999), Lim, *et al.* (1994), and Li, *et al* (1996), analyse cache behaviour to estimate task execution time considering the intra-task interference. Busquets, *et al* (1996), and Lee, *et al.* (1997), analyse cache behaviour to estimate task response time considering the inter-task interference, using a precalculated cached WCET. Kirk (1989), Liedtke, *et al.* (1997), and Wolfe (1993),

[1]This work was supported in part by *Comisión Interministerial de Ciencia y Tecnología* under project CICYT-TAP 990443-C05-02

use hardware and software techniques to eliminate or reduce the inter-task interference.

The main drawback of previous solutions is that they only solve one side of the problem: intrinsic interference or extrinsic interference, and they consider that the other side is already solved by a different technique. Furthermore, in order to get accurate response time estimations, complex analysis techniques are required.

The main goal of this work is to present a cache scheme capable of offering full predictability for WCET estimation, and a bounded value of extrinsic interference, required for the schedulability analysis. This goal is achieved using the cache-managing instructions that are present in modern processors, instructions such as selective preload (cache fill) and cache locking. These features are used in order to attain: easy schedulability analysis, accurate WCET estimations, and high performance (actual execution times similar to running with conventional and unpredictable cache).

The technique is based on the ability of several processors in locking down the cache, precluding the removal of its contents, but allowing references to the data or instructions already stored within the cache. When a task begins or resumes its execution, a preselected set of instructions is loaded and locked in the cache. Thus, intra-task interference is eliminated, since cache content remains unchanged during task execution. After preemption, a task always reloads the same set of instructions, and as a result, the value of the cache refill penalty is fixed and known.

This work only considers instruction cache, without regard to other architecture improvements. The paper is organised as follows: the following section illustrates the hardware and system operation required to reach both predictability and the best possible performance. Section three introduces the algorithms for calculating WCET and Response Time when a locking cache is used. Next, a description of the genetic algorithm used to select the best set of instructions to load in a cache is presented. Finally, experimental results are described.

2. SYSTEM OVERVIEW

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

The IDT-79R4650 cache schema offers an 8KB two-set associative instruction cache. Also, the processor offers the instruction “cache fill”, an instruction to selectively load cache contents. However, this processor allows locking a single cache set, leaving the remaining cache set unlocked Since the main objective of this work is to reach a deterministic cache, locking the entire cache is required. In the MPC7400, it is possible to lock the entire cache, using a one-line size buffer to temporally store instructions not loaded in the cache, consequently improving sequential access to these addresses. The problem with this processor is that no selective load of cache contents is available. Therefore, this work proposes a merge of the two above-mentioned processors, resulting in a cache system with the following characteristics:

- The cache can be totally locked or unlocked. When the cache is locked, there are no new tag allocations.
- If the processor addresses an instruction located in the locking cache, this instruction is served from the cache.
- If the processor addresses an instruction located in the temporal buffer, this instruction is served from this buffer within cache access time.
- If the processor addresses an instruction that is not located in the locking cache or temporal buffer, this instruction is served from main memory. The temporal buffer is filled with the block containing the address demanded by the processor.
- The cache can be loaded using a cache-fill instruction, selecting the memory block to be loaded.
- The cache can be locked, unlocked and flushed using cache-managing instructions.
- The cache may be a direct mapped cache or a set associative cache. Increasing the associative-level may increase performance, but a direct-mapped cache is enough to reach predictability.

A fully locked cache allows obtaining the maximum possible performance, while making the cache deterministic. The temporal buffer reduces access time to the memory blocks that are not loaded in the cache, since only references to the first instruction in the block produce cache miss.

For each task, a set of instructions will be selected to be locked in the instruction cache. The address of each main memory block is stored in a table called Task’s Locking Table (TLT). This TLT is filled by a post-processing tool after the executable file is generated. The operation of the locking-cache system is as follows:

- Before executing a task, the OS loader invalidates the cache (clearing all locked areas) and loads and locks the instructions referenced in the TLT.
- After preemption, and just before the resumed task is executed, the OS scheduler invalidates the cache, and loads and locks the instructions referenced in the TLT.

This way, when a task begins or resumes its execution, the cache contents are a-priori known, and remain without change until the task ends or is preempted. Thus, cache behaviour is now predictable. Preloaded instructions can belong to any part of the task, and may

be large consecutive instruction sequences or small, individual separate blocks.

3. SCHEDULABILITY ANALYSIS

Schedulability analysis may be achieved using 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, which may be compared to the task deadline. In this equation, W_i denotes the response time of task τ_i, C_i is the WCET of τ_i without preemptions, Bi denotes the time task τ_i is blocked, T_j is the period of task τ_j and hp(i) is the set of tasks with higher priority than task τ_i.

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

In a cached system, the schedulability analysis must consider the effect of the cache, and RTA must be extended to take account of the cache refill penalty. CRTA (Busquets, *et al.* 1996) may be used to calculate the response time of each task when cache memory is present. Equation 2 shows the expression of CRTA, where C_i is the WCET of τ_i without preemptions but considering cache effect, and γ_j is the rise in the response time that task τ_i experiences due to task τ_j.

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

In a preemptive system and regarding cache, a task may suffer two types of extrinsic interference: direct interference or indirect interference. Direct interference means that a task increases its response time because it is forced to reload its own instructions that had been previously removed during preemption. Indirect interference means that a task increases its response time because other higher priority task increases its response time, due to its own extrinsic interference. In Figure 1.a, task 3 increases its response time because it is forced to reload cache contents after executions of task 2 and task 1. In this case, task 3 only suffers direct interference. In Figure 1.b, task 2 increases its response time by reloading cache contents after the execution of task 1 (direct interference), and task 3 increases its response time by reloading cache contents after the execution of task 2 (direct interference) and due to the increment in the response time of task 2 (indirect interference). In both cases, task 3 is preempted twice, but the value of the cache refill penalty is different for each scenario.

The value of direct-extrinsic interference is the time a task needs to load and lock its instructions in the cache. The value of indirect-extrinsic interference is the time other higher priority task needs to load and lock its instructions in the cache. Since response time analysis must consider de worst case scenario in order to provide an upper bound of tasks' response time, the maximum possible increment of time must be considered for each preemption in the CRTA expression. This way, a task τ_i, preempted by a task τ_j, can increase its response time due to its direct interference, or due to the increment suffered by other tasks (indirect interference), with higher priority than task τ_i and lower priority than task τ_j. The higher value obtained will be the cache refill penalty. Equation 3 shows the cache refill penalty expression for a task τ_i preempted by a task τ_j; and equation 4 shows the CRTA expression using the proposed locking-cache scheme.

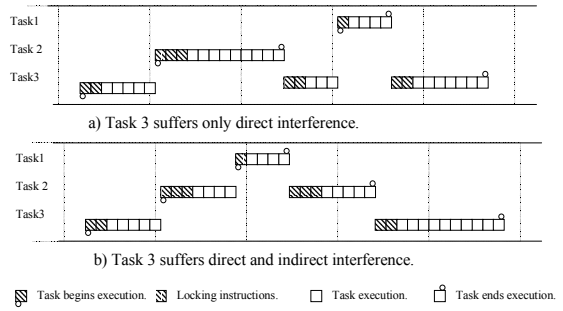

Figure 1. Example of direct and indirect extrinsic interference.

$$\gamma_j^i = \underset{j<z\leq i}{MAX}(time\_to\_load_z) \quad (3)$$

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil x (C_j + \gamma_j^i) \quad (4)$$

In the CRTA expression, C_i represents the WCET of task τi, and this must be calculated while taking into account the existence of the cache. To calculate the WCET of a task and taking into account the presence of the locking cache, a modified timing analysis (Shaw, 1989) is proposed.

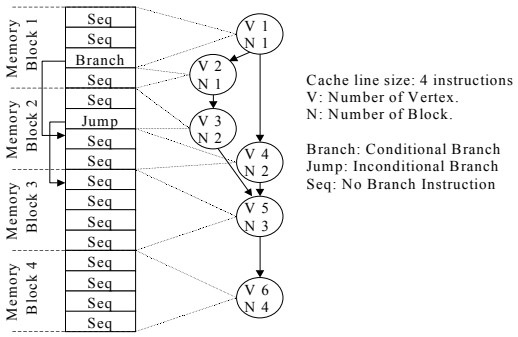

Figure 2. Example of c-cfg.

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 vertex meaning, since the c-cfg models not only the task's paths but also how the cache is used. Figure 2 illustrates an example. This c-cfg can be represented with a simple string, an expression that can be

evaluated to obtain the WCET task. Figure 3 shows the expression for three basic c-cfg. In these expressions, E_i represents the execution time of vertex V_i.

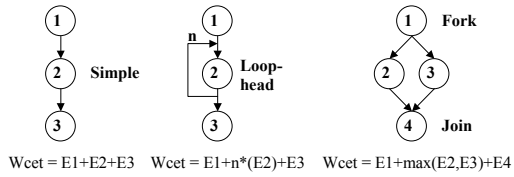

Figure 3. Expressions for three basic structures.

The WCET of a task can be calculated by evaluating the expression while taking into account the execution time of each vertex. The execution time of a vertex depends on the number of instructions into the vertex and on 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 remains 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 the cache, its execution time E_i is: $E_i = T_{hit} * I_i$
- For a vertex V_i not loaded in the 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 the cache, and T_{miss} is the time to transfer a block from main memory to the temporal buffer. Vertex execution time can be directly used in the c-cfg expression in order to obtain the WCET of the task, obtaining an upper bound value, since execution time is now independent of cache.

Since each task must load and lock the selected instructions before it can begin execution, the time needed to load and lock cache contents must be added to the WCET of the task. This time, called time_to_load, is machine dependent, and is calculated in the experimental results section.

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

Randomly loading and locking instructions in the cache offers predictability but does not guarantee good task response time. In order to reach both goals, a predictable cache and a like-cache performance, instructions to be loaded must be carefully selected, in search for the best scenario. This scenario is a set of main memory blocks locked in the cache that provides the minimum possible execution time, thus providing the minimum possible response time for a set of tasks.

Genetic algorithms (Goldberg, 1989), performing a randomly-directed search, can be used in this problem, finding a sub-optimal solution within an acceptable computational time. The proposed algorithm provides the set of main memory blocks, an estimation of the WCET of each task executed in a locked cache with the set of blocks loaded and locked, and the response time of all tasks taking into account the WCET estimated using the locking cache. The genetic algorithm used in this work is the evolution of a previous version presented by Martí, *et al.* (2001). The main characteristics of the new algorithm are described below:

Each block can be locked or not in the cache. An individual represents, in a single chromosome, the state of all blocks of all tasks in the system, 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 task considering the state – locked or not- of the blocks. Task response time is calculated using the CRTA and the WCET expressions described in previous section.

The existence of invalid individuals (number of locked blocks greater than cache size) precludes the use of direct probability setting as function of fitness value in order to perform crossover. This way, individuals are arranged according to their validity degree, considering both the number of locked blocks and the fitness 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 for crossover, with low probability, non-valid individuals that help to increase the variability of the algorithm.

Crossover is performed by randomly choosing a gene that divides the individual into two parts, and by exchanging the parts of two individuals, creating two new individuals. Mutation, on a gene-basis, is applied to each new individual, and is applied for each task into the individual. Also, mutation may exchange locked blocks in order to guarantee that a direct-mapped locking cache is used. A new population is created with the individuals obtained from crossover and mutation, and the process is repeated for a previously defined number of times. For the accomplished experiments presented further in this paper, the number of iterations is established in 5000, the population is formed with 200 individuals, crossover probability is 0.6, and mutation probability is established in 0.001.

The genetic algorithm simultaneously solve the problem of block selection and schedulability analysis, since the response time values obtained from the genetic algorithm are an upper bound of the task response time, and can therefore be compared with task deadlines to validate the system schedulability.

5. EXPERIMENTAL RESULTS

Experimental results must show if the proposed use of locking caches makes the system predictable, and if the performance loss (if any) is reasonable. To make experiments, the SPIM tool (Patterson and Hennessy,

1994), 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 the instructions of each task is also incorporated in the simulator, but its execution time is added to the calling task. The assembly code for this routine is shown in Figure 4. The routine is never loaded in the cache, but profits from the temporal buffer. The execution time of this routine is estimated in 46 cycles for each block to be loaded (under terms described further), plus a constant of 12 cycles. This value is accounted in the genetic algorithm during WCET and Response Time estimation.

```
flush $0,$0,$0      # invalidate and unlock the cache
ini:  addi $1,$1,-1 # assume register 1 contains number of
                    blocks to be locked
lw $3,0($2)         # assume register 2 contains initial
                    address of TLT
lab $3              # Load memory block addressed by
                    register 3
addi $2,$2,4        # Point to the next TLT entry
bne $0,$1,ini
nop
lock                # Lock the cache
jr $31              # Return from routine
```

Figure 4. Load and lock function for a MIPS R2000-like processor

Tasks used in experiments are artificially created to stress the proposed cache scheme. Main parameters of task are defined, such as the number of loops and nesting level, the size of tasks, the size of loops, the number of if-then-else structures and their respective sizes. A simple tool is used to create tasks. Task period is hand-defined to make the system schedulable, and the 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 (up to 64 Kb) or short (lower than 1Kb).

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 full associative cache, with cache sizes 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 blocks selected by the genetic algorithm.

Figure 5 presents the error between the response time of every task; either estimated by the genetic algorithm or simulated using a locking cache. Experiments collect more than 700 executions. Each bar represents the number of tasks with an estimated error that lies in the interval of the x-axis. This figure shows that the proposed technique is conservative, since the estimated response time is always larger than the simulated one. Figure 6 shows the accumulated frequency: accumulated number of tasks for the given error between simulated and estimated response time using a locking cache. Axis-y value is the percentage of tasks with an error lower than axis-x value. It can be observed that the estimated response time is tight: around 60% of the cases present an error below 2%, and 90% of them present an error below 9%.

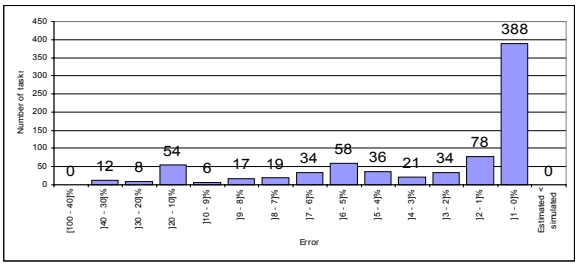

Figure 5. Error between simulated and estimated task response time using a locking cache.

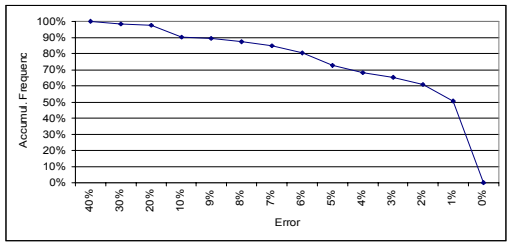

Figure 6. Accumulated frequency of error between estimated and simulated response time.

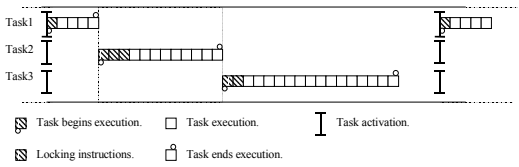

Figure 7. Example of preemption without cache refill penalty.

Error between estimated and simulated response time is mainly due to the conservative approach of CRTA. CRTA equation considers that all preemptions occur during task execution, since this is the worst case scenario. This way, the time to reload cache contents is added to the response time for each possible preemption. But in fact, a task may suffer a preemption after activation and before execution. In this case, no time to reload cache contents is needed, but the CRTA equation does not differentiate between these preemptions. Figure 7 illustrates an example where three tasks are activated within the same time instant. Task 3 suffers two preemptions but does not need to reload cache contents, since its execution has not yet begun.

Regarding the performance of the locking cache, Figure 8 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 according to this ratio. Figure 9 draws the accumulative values of the previous figure. For around 45% of the tasks, the response time is equal or better using locking cache, and near of 70% present a performance ratio over 0,9.

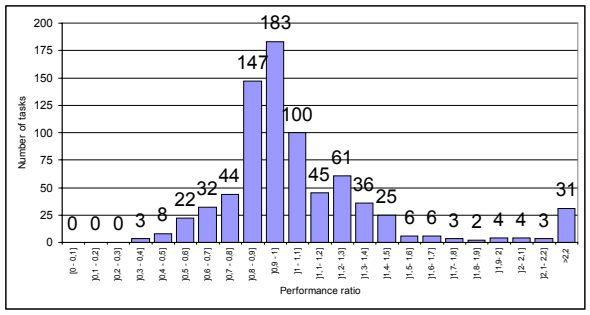

Figure 8. Task performance ratio obtained when using locking cache. Each bar represents the number of tasks with performance ratio that lies in the interval of the x-axis.

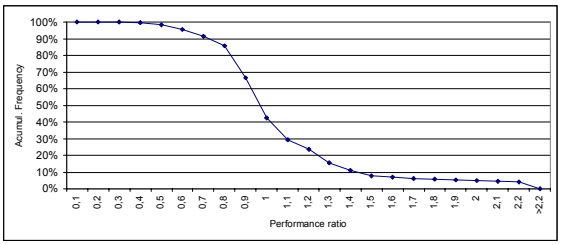

Figure 9. Accumulative task performance ratio when using locking cache. Axis-y value is the percentage of tasks with performance ratio greater than axis-x value

6. CONCLUSIONS

This work presents a novel technique that uses locking caches in the context of real-time systems. Furthermore, algorithms to analyse the proposed system are described. Compared to other known techniques used to achieve cache predictability in these systems, this solution completely eliminates intrinsic cache interference, and gives a bounded value of extrinsic cache interference, providing an accurate response time estimation. This predictability is reached with no loss of performance for around 70% of the experiments.

The benefits of a 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 used to improve performance, such as segmentation, thus precluding the consideration of the complex interrelations among these techniques and the cache.
The hardware resources required to implement this scheme are available in some contemporary processors. To obtain the best results, some minor changes might be introduced. These changes do not present difficulties in terms of technical complexity and production.

An algorithm to select cache contents is presented. This selection delivers the best performance. The algorithm also calculates each task's WCET and response time.

REFERENCES

Busquets, J. V. J. J. Serrano, R. Ors, P. Gil, A. Wellings (1996). Adding Instruction Cache Effect to an Exact Schedulability Analysis of Preemptive Real-Time Systems. *Proc. of the IEEE Euromicro Workshop on Real-Time Systems*

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization and machine Learning Addison-Wesely Co.

Healy, C. A., R. D. Arnold, F. Mueller, D. Whalley and M. G. Harmon (1999). Bounding Pipeline and Instruction Cache Performance. *IEEE Transaction on Computers*. **Volume 48**, pages 53-70

Kirk, D. B. (1989) SMART (Strategic Memory Allocation for Real-Time) Cache Design. *Proc. of the 10th IEEE Real-Time Systems Symposium*.

Lee, C. G., J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, C. S. Kim (1997). Enhaced Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. *Proc. of the 18th IEEE Real-Time System Symposium*.

Li, Y. S., S. Malik, and A. Wolfe (1996). Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. *Proc. of the 17th IEEE Real-Time Systems Symposium.*

Liedtke, J., H. Härtig, M. Hohmuth (1997). OS-Controlled Cache Predictability for Real-time Systems. *Proc of the IEEE Real-Time Technology and Applications Symposium*.

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 (1994). An Accurate Worst Case Timing Analysis Technique for RISC Processors. *Proc. of the 15th IEEE Real-Time Systems Symposium*.

Martí, A., A. Pérez, A. Perles, J.V. Busquets (2001). Using Genetics Algorithms in Content Selection for Locking- Caches. *IAESTED International Conference on Applied Informatics.*

Patterson, D. Patterson and J. L. Hennessy (1994). Computer Organization and Design. The Hardware/Software Interface. Morgan Kaufmann. San Mateo.

Shaw, A. (1989).Reasoning About Time in Higher-Level Language Software. *IEEE Transaction on Software Engineering.* **Vol. 15**, Num. 7.

Wolfe, A. (1993). Software-Based Cache Partitioning for real-time Applications. *Proc of the 3th International Workshop on Responsive Computer Systems.*