"Combined Intrinsic-Extrinsic Cache Analysis for Preemptive Real-Time Systems"
A. Martí Campoy, X. Molero, A. Perles, F. Rodríguez and J.V. Busquets.
Real-Time Programming 2000. Proceedings of 25th IFAC Workshop on Real-Time Programming. pp 49-55. Pergamon, 2000

COMBINED INTRINSIC-EXTRINSIC CACHE ANALYSIS FOR PREEMPTIVE REAL-TIME SYSTEMS

A. Mart, X. Molero, A. Perles, F. Rodrguez, and J.V. Busquets

Departament d'Informtica de Sistemes i Computadors
Universitat Politcnica de Valncia, Spain
email: amarti@disca.upv.es

Abstract: Cache memories are widely used to improve computer performance, but their inherent unpredictability presents new problems when cached systems must be analysed. In preemptive, multitask real-time systems, the cache memories have been analysed from two complementary points of view. First, calculating the Worst Case Execution Time (WCET) of each task considering no preemptions. Second, making the schedulability analysis considering the effect of cache when tasks are preempted. Both aspects of the same problem (performance variation due to cache) have been historically treated independently. This paper presents a new approach to deal with both problems altogether when a direct mapped cache is used for instructions. Solving both problems jointly provides greatter accuracy in the schedulability analysis.
Copyright © 2000IFAC

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

1. INTRODUCTION

Caches greatly contribute to increase computer performance. But the use of cache memories in preemptive real-time systems presents two problems because of their unpredictable behaviour. The first problem is to calculate the Worst-Case Execution Time (WCET) of cached tasks, due to intra-task interference. The second problem is to calculate preemption cost due to the inter-task cache interference. Intra-task or intrinsic interference arises when a task removes its own instructions from cache. When removed instructions are executed again, a cache miss increases the execution time of the task. Inter-task or extrinsic interference arises in preemtive, multitask systems when a task removes another task from cache. When the preempted task resumes execution a burst of cache misses increases its response time.

During the past few years, several proposals have been presented to obtain the WCET of tasks taking into account the cache speed-up (Mueller and Wegener, 1998; Lim et al., 1994; Li et al., 1996). These techniques assume that the task under analysis executes without preemption, because from the point of view of program analysis, the preemption point is unknown. Other papers have addressed the extrinsic cache interference problem in the schedulability analysis in preemptive systems, extending non-cached analysis (Busquets et al., 1996; Basumallick and Nilsen, 1994; Lee et al., 1997). These calculate the increment in the task response time (called cache refill penalty or cache-related preemption delay) and add this time to the pre-calculated WCET. The importance of the cache refill penalty is shown in (Lee et al., 1999), where selecting the preemption point with minimal inter-task interference improves schedulable utilization by up to 40%. None of the techniques to compute the extrinsic interference obtains valu-

[1] This work is supported by project CICYT TAP990443-C05-02

able information from the WCET analysis except (Kastner and Thesing, 1999), where cache analysis is made in each preemption point. However, (Lee et al., 1999; Kastner and Thesing, 1999) work with non preemtive or limited preemptive systems. We argue in this paper that the cache refill penalty can be obtained while obtaining the WCET for the task, and that the resulting value is more accurate when calculated in this way for preemptive real-time systems.

Figure 1 shows an example of a simple task with four blocks (a block is the minimum unit of information that can be either present or not in the cache-main memory hierarchy) using a direct-mapped cache. Instructions in blocks 2 and 3 compete for the same cache line, C1. In the WCET analysis it is considered that execution of blocks 2 and 3 will always produce cache misses due to the possibility of alternate paths while executing the loop. Analysing the inter-task interference for the schedulability analysis, the time to reload the contents of cache line 1 must be considered after a preemption, because it is possible that the task executes the loop using the same path repeatedly. If no other information is used from the WCET analysis but the value of the task WCET, the miss penalty of cache line 1 is counted twice in the global analysis: in the task WCET and in the cache refill penalty. However, the displacement of instructions from cache line 1 after a preemption does not increase the task response time over the pre-calculated WCET (as it has been considered that addresing cache line 1 will always produce a cache miss).

To avoid this overestimation, this paper presents a novel approach to offer a unique and coherent solution to both problems. The proposed approach has been developed with simplicity in mind for its use in real applications. This solution uses a set of simple and fast algorithms to obtain an upper-bound of the WCET of cached tasks considering intra-task interference. In addition (and what makes it different to other approaches) it also provides a coherent estimation of the preemption cost to be considered in the schedulability analysis to cope with the inter-task interference, offering a global solution to the problem when a direct mapped cache is used for instructions. As a first approach, this work has been focused only on instruction caches. On average, four out of five memory accesses are instruction fetches (Hennessy and Patterson, 1996), thus instruction cache has greater influence on the processor performance.

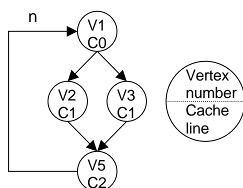

Fig. 1. An example of overestimation

2. WCET CALCULATION

The first step to make the schedulability analysis is to calculate the WCET of each task. In this section, algorithms to calculate the WCET of a cached task are presented. The method to obtain the WCET is divided into three parts: expression, categorisation, and resolution. First, an expression to calculate the task WCET is obtained. Second, the execution time of each vertex is obtained, as a function of the cache state. Finally, evaluating the expression using the vertex execution time provides the task's WCET. The algorithms presented fulfill into two goals: first, they allow a fast and easy analysis, and second, they provide useful information to calculate the cache refill penalty.

2.1 *Expression*

From the task's Control Flow Graph and machine code, a new Control Flow Graph, called cached-cfg (c-cfg), is created following these rules:

- A vertex is a sequence of instructions without flow break.
- All instructions on a vertex map in the same cache line.
- A vertex can appear only once in the c-cfg.

This model differs from conventional CFG in the meaning of a vertex, since in this model the vertex models not only the task's paths but also how the cache is used. Figure 2 illustrates an example.

Function calls in the c-cfg (that violate the third condition above) are replaced by the c-cfg of the function body, renaming the vertexes to make them unique but maintaining the block number (as if the function were compiled as an 'inline' function). For each vertex the following information is stored:

- The execution time of each vertex in case of cache miss (Tm) and in case of cache hit (Tc).
- The block number for each vertex.
- The cache line number where the vertex maps.
- If the vertex is a loop begin, maximum number of iterations (user provided).
- Number of edges that reach the vertex, without counting closing-loop edges (join field).

Restrictions imposed to tasks are (See figure 3):

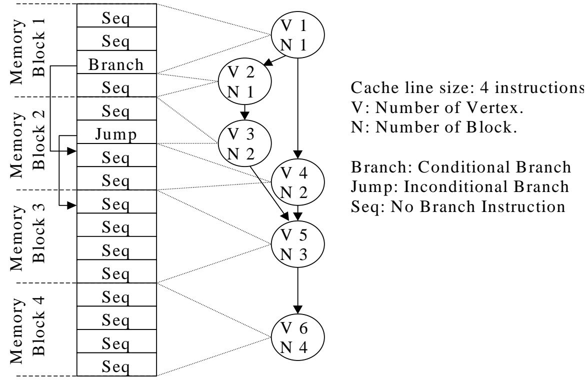

Fig. 2. Cache-Control Flow Graph example

- The task begins in a single vertex and ends in a single vertex.
- Loops have a unique exit vertex.
- Only two edges can depart from a vertex.
- No recursive function calls are allowed.
- Only simple program constructs are allowed in the task cfg.

These programming structures allow the task cfg to be represented with a simple string, an expression that can be evaluated to obtain the task WCET. Though the last restriction seems too strict, complex control structures can be rewritten using a combination of these. This can be accomplished using specialized compilers, or with a post-processing tool that rewrites the task cfg. Figure 3 shows the WCET expression for the allowed task structures. In the WCET expression, Ei represents the execution cost of vertex i, which will depend on its instructions and the cache state.

The algorithm recursively traverses the c-cfg updating the expression depending on the vertex type (simple, join, fork or loop-head) as depicted in figure 3. When the algorithm finds a "special" vertex (join, fork or loop-head), it updates the expression adding the corresponding string to the WCET expression, eliminates the vertex attribute and reprocesses it. In this way, the special vertex is gradually transformed into a simple vertex, finally the algorithm adds the vertex number to the expression. Simultaneously, if a vertex is inside a loop, this vertex is marked with the number of the loop-head vertex. For nested loops, the vertex is marked with all loop-head vertexes. Algorithm cost is $O(n)$, n being the number of vertexes in the c-cfg. The main characteristic of the expression obtained is that each vertex appears only once.

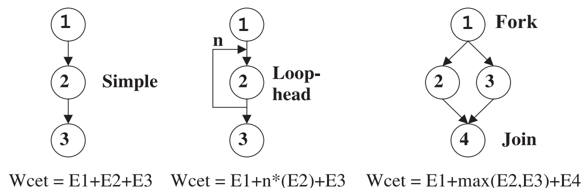

Fig. 3. WCET expressions and vertex types

Table 1. C-cfg of Figure 4 task

Vertex	Block num.	Edge 1/2	Iterations	Join	Cache line	Mark
1	1	2/0	1	1	0	0
2	2	3/0	10	1	1	2
3	2	4/0	15	1	1	2,3
4	4	5/6	1	1	2	2,3
5	4	7/0	1	1	0	2,3
6	6	7/0	1	1	0	2,3
7	7	8/3	1	2	0	2,3
8	8	9/0	1	1	1	2
9	8	0/0	1	1	0	0

This will allow a unified treatment of compulsory and conflict misses (Hennessy and Patterson, 1996), as well as simplifying the cached task WCET calculation.

Figure 4 presents the c-cfg of a sample task. All the information stored for each vertex is contained in table 1. The mark field indicates the loop-head vertex for each loop. The cache lines for each vertex have been manually selected for subsequent examples.

2.2 *Categorisation*

To evaluate the WCET expression obtained in the previous section, it is necessary to know the execution time of each vertex. In absence of a cache, the execution time of a vertex is the sum of the execution times of each of its instructions. However, when a cache is present, this cost can vary in each execution. In this section, a categorisation of each vertex is presented, that will indicate if the vertex is in cache each time it is executed, thus indicating its execution time. Mueller et al. (1998) use a categorization to calculate WCET when cache memory is used, creating an accurate and precise categorisation, allowing a more accurate WCET. However, the principal objective of this paper is not only to calculate an accurate WCET, but also to obtain the necessary information to calculate the cache refill penalty produced by task preemptions. In this way, the authors have preferred to simplify the categorisation, introducing an overestimation in the obtained WCET, but that potentially reduces the cache reload cost after preemptions. In this way, three vertex types are defined:

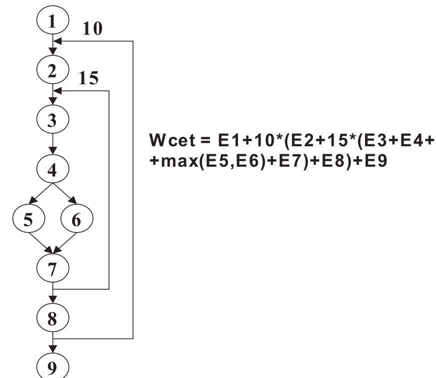

Fig. 4. WCET expression of an example task

Unique: it is the only vertex mapped onto a particular cache line. Therefore, during the execution time of the task only the first time the vertex code is executed will produce a cache miss.

Shared: two or more vertexes map onto the same cache line but they are mutually exclusive. Once one vertex is replaced from the cache, it will never be executed again. Like the former type, this vertex type suffers from cache miss only in its first execution.

Conflict: two or more vertexes with different block number map onto the same cache line, and a vertex replaced from cache can be executed again. Vertexes of this type can suffer an indeterminate number of cache misses.

The algorithm to categorise c-cfg vertexes is easy: a vertex that maps alone in a cache line is Unique. A vertex a that maps in the same cache line that vertex b (a and b with different block number) is classified as Conflict if paths exist to reach a from b, and to reach b from a. Otherwise, a is Shared. As the computational cost to determine the existence of a path from a to b and vice-versa is very high, the conflict condition is redefined as follows: a pair of vertexes a and b conflict if and only if both are marked with the same loop-head, because if vertexes a and b are inside the same loop, then there is a path from a to b and viceversa.

However, for nested loops it is necessary to identify the dependencies between different nest levels. In this way, a vertex will be categorised for each loop that it belongs to, and the classification of a vertex at a given level will depend exclusively on vertexes existing at the same level. This includes the vertexes in the same loop and innermost loops, but not those in the outermost. In this way the categorisation takes into account the dependencies created between vertexes belonging to different loops.

For each vertex and for each nesting level, the algorithm searches for another vertex mapping to the same cache line and with different block number. If such a vertex exists, and belongs to the same or innermost nesting level, both vertexes are classified as Conflict for this nesting level. Otherwise, the vertex will be classified as Unique or Shared.

In the worst case, all vertexes map in the same cache line, and they are at the same nesting level. In this particular case, $l \times n \times (n-1)$ comparisons are needed, n being the c-cfg vertex number and l the maximum nesting level. These comparisons do not need to traverse the c-cfg, as the mark field directly indicates the existence of a path between any two vertexes in the c-cfg, and easy optimizations are possible. Table 2 presents the result of applying the categorisation algorithm to the c-cfg of the figure 4.

Table 2. C-cfg of Figure 4 task categorised

Vertex	Categorisation
1	Shared
2	Conflict2
3	Conflict2
	Shared3
4	Unique
5	Conflict2
	Conflict3
6	Conflict2
	Conflict3
7	Conflcit2
	Conflict3
8	Conflict2
9	Shared

2.3 *Resolution*

Once the categorisation for each vertex is obtained, it only remains to evaluate the expression, using the execution time of each vertex as a function of its categorisation. However, in a system with a cache, the execution cost of such a vertex will depend on its categorisation for each nesting level. Therefore, the execution time of a vertex inside a nested loop will be calculated independently for each level using the vertex categorisation for that level, beginning in the innermost loop, and combining the obtained times.

The WCET expression obtained guarantees that a vertex will appear only once, which allows the calculation of the WCET to be simplified by dividing the process into the following cases:

The vertex is outside a loop: this vertex will be executed only once, therefore this occurrence is the first and last. Independently of its categorisation, the execution time is Tm.

The vertex is within a single loop of n iterations. In this case, the vertex will be executed n times and the execution time will depend on its categorisation:

- Unique: the first time this vertex is executed produces a compulsory miss, and will stay in cache while the task is executed. Execution time is: $Tm + (n-1) \times Tc$.
- Shared: the first time this vertex is executed produces a compulsory miss and will stay in cache during the following $n-1$ executions, producing hits. It can be replaced from cache, but it will not execute after this. Execution time is again: $Tm + (n-1) \times Tc$.
- Conflict: it is not possible to guarantee that the vertex will stay in cache after each execution. Therefore each execution must be considered as a cache miss, taking: $n \times Tm$.

The vertex is inside z nested loops, with iterations $n_1, n_2, \ldots, n_z$. The combinations of the vertex categorisation at each level (considering 1 as the out-

ermost and z as the innermost levels, respectively) are given below:

- All unique: if this categorisation is applied for every loop, the execution time is $Tm + ((n_1 \times n_2 \times \ldots \times n_z) - 1) \times Tc$.
- All shared: in this case, the vertex can be replaced from cache if it is no longer used, so the execution time is $Tm + ((n_1 \times n_2 \times \ldots \times n_z) - 1) \times Tc$.
- All conflict: in this simple case, the execution time is $n_1 \times n_2 \times \ldots \times n_z \times Tm$.
- Conflict 1 - ... - conflict k - shared k+1 - ... - shared z: the cache misses for the vertex can be produced in the 1, ..., k outermost loops only. Every time the execution flow exits from the $k+1$, ..., z innermost levels, a cache miss must be counted, so the time is $n_1 \times \ldots \times n_k \times Tm + n_1 \times \ldots \times n_k \times ((n_{k+1} \times \ldots \times n_z) - 1) \times Tc$.
- Other combinatios are not possible, due to the fact that when a vertex is categorised as conflict in an nested loop, it is assumed that it will be replaced from cache every time it is executed. As each vertex can appear only once, it is not possible to consider the vertex as shared at the outer levels.

The cost of the algorithm is $O(n)$, where n is number of vertexes in the c-cfg.

3. SCHEDULABILITY ANALYSIS

To carry out the schedulability analysis of cached, preemptive real-time systems, the equation of Cache Response Time Analysis (CRTA) presented by Busquets (Busquets et al., 1996) is used:

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

where w_i denotes the response time of task τ_i, C_i the WCET of τ_i considering the presence of a cache, B_i denotes the task blocking time, T_j is the period of task τ_j and γ_j indicates the cache refill penalty added to task τ_i for each release of task τ_j. The set $hp(i)$ is the set of tasks with higher priority than τ_i.

From the CRTA equation, all values can be easily obtained except the cache refill penalty for each task, γ_j. To obtain this value it is necessary to know the number of useful lines of each task, defining a line as useful when the block loaded in this line executes two or more times before it is replaced from cache. This analysis must be performed with a conservative criterion. If it is not possible to decide exactly if a line is useful, it must be considered as useful to guarantee an upperbound of the cache-related preemption delay. However, when calculating the WCET, the conservative criterion is the opposite: if it is not possible to decide exactly if a line is useful, it must be considered as not useful (cache miss) to guarantee an upperbound of the WCET. If we call U_w the number of useful lines for the WCET, and we call U_a the number of useful lines that will be considered in the schedulability analysis, we usually obtain $U_a > U_w$, what introduces a double conservatism as shown in the introduction.

To reduce this double conservatism, the useful lines of a task are obtained from the algorithms previously presented. In this way, the number of useful lines is the number of different cache lines which vertexes map into, provided that they are categorised as Unique or Shared and they are inside a loop. Only the displacement of these lines from cache after a preemption increases the response time of the task, thus only these lines must be considered as useful. However, not all of these lines are actually useful. A multiple-categorised vertex (a vertex inside a nested loop), produces a useful line if at least one of its categories is Shared or Unique.

In this way, the number of useful lines can be trivially obtained from the categorisation and the mark field using the first algorithm. In the example of figure 4, vertexes that produce useful lines are 3 and 4 (1 and 9 are not inside a loop), and these vertexes map in cache lines 1 and 2, therefore the number of useful lines is 2.

This information (the number of useful lines of each task) can be applied in the equation of the CRTA in several ways. A linear programming technique can be used, as explained in (Lee et al., 1997) or the process described in the next paragraph can be applied. The first method is more accurate, the second is easier and faster.

The cache-related preemption delay suffered by a task depends on the direct interference (useful lines replaced by the preempting task), and the indirect interference (the preempting task increases its response time because its useful lines are replaced from cache by a higher priority task). The value of γ_i will be the maximum number of useful lines that the task τ_i or a higher priority task (except the highest) must reload after a preemption. This results in the following equations:

$$\gamma_i = \max_{j \in hp(i)-1} (useful_j) \quad (2)$$

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times (C_j + \gamma_i) \quad (3)$$

This value is an upperbound of the cache-related preemption delay, and it is coherent with the execution times calculated for the tasks. This guarantees valid results and precludes double overesti-

mation. In addition, the complete process is easy and it has a low computational cost.

4. EXPERIMENTAL RESULTS

The algorithms presented to calculate the WCET of cached tasks have been implemented in an application created in C++ and executed on a mid-range personal computer. Two kinds of experiments have been performed: experiments to show the computational time of algorithms and experiments to evaluate the accuracy of the estimated WCET.

To evaluate the computational time of algorithms, several c-cfgs not representing real tasks, with different number of vertexes and cache sizes have been processed. Results are shown in table 3. For all cases, response time of algorithms are lower than one minute.

The accuracy of the estimated WCET is evaluated by processing four small functions:

- Task1: a task that calculates the sum of the first one hundred numbers.
- Task2: a task that sequentially searches for the maximum element in a one hundred elements array.
- Task3: a task with two nested loops, with 10 iterations each one, calculating the sum of innermost counter.
- Task4: the same as task3, but the number of iteration of the inner loop depends on the counter of the outermost loop.

These tasks were written in C, compiled with gnu gcc compiler and disassembled on a Silicon Graphics workstation, with MIPS R5000 and Irix OS. The actual WCET of different tasks has been calculated using a modified version of SPIM (Patterson and Hennessy, 1994), a simulator for MIPS RISC processors. The main modification consists in adding an array that represents the cache tag. Executing an instruction from main memory costs 11 units, and executing an instruction from cache costs 1 unit.

The cache-simulated WCET has been obtained using the WCET expression and a table where the instruction address loaded in each cache line is maintained. With this information, the WCET expression is recalculated for each loop iteration,

Table 3. Examples of calculated c-cfgs

c-cfg size (vertexes)	Cache size (lines)	WCET expr (sec)	Vertex cat (sec)	WCET calc (sec)
10	1	1	1	1
100	1	1	9	1
100	10	1	1	1
1000	100	6	17	2
1000	500	6	7	7
2000	500	18	24	7
2000	1000	18	19	8

Table 4. Task execution cycles

Task	Cache lines	iterations	Spim sim.	Expr sim.	Estimated	Error
Task1	2	100	2952	2952	2962	0,34%
Task1	4	100	972	972	982	1,03%
Task2	4	100	8237	8237	8347	0,24%
Task2	4	500	41527	41527	41547	0,05%
Task3	2	10x10	3333	3333	3453	3,6%
Task3	4	10x10	1263	1263	1293	2,3%
Task3	4	20x20	4253	4253	4283	0,7%
Task4	4	10x10	1153	1563	1693	49,1% /8,3%
Task4	4	20x20	3185	5063	5293	66,1% /4,5%

considering the cache contents (not the vertex categorisation). Finally, the estimated WCET has been calculated using the WCET expression, the vertex categorisation and the procedure to solve the expression shown in section 2.

Results are shown in table 4. For task1, task2 and task3, results from SPIM simulation and expression simulation confirm that the WCET expression actually represents the task WCET. In task4, the error in the simulated expression is due to an erroneous specification of the maximum number of iterations of the inner loop. This number of iterations is statically specified, while the actual task has a dynamic value that depends on the outer loop counter. Table 4 shows, for task4, the overestimation with respect the actual WCET due to loops and cache, and with respect to expression simulation due to the cache only. Regarding the estimated WCET, a small overestimation is obtained due mainly to simplified categorisation of divided blocks.

5. CONCLUSIONS AND FUTURE WORK

This paper has presented a unified and coherent solution for both aspects to be considered when using a cache in preemptive real-time systems: intrinsic and extrinsic interference. Historically, they were treated as two separate and independent problems: the intra-task cache interference in the WCET estimation, and the inter-task interference in the schedulability analysis.

The work carried out obtains the WCET from the task Control Flow Graph and the machine code, and also obtains the necessary information to perform the schedulability analysis considering the cache effect. Three easy and practical algorithms obtain this information. The calculated WCET maybe worse than the one obtained with other methods, but it may offer a more accurate response time when calculating the schedulability analyisis.

The future and ongoing work has three lines: the refinement of WCET calculation, without excessive increase in response time, the refinement of cache-related preemption delay (γ_i) calculation for the CRTA equation, and tunning: the ultimate

objective is to obtain a response to the question: is my system schedulable? A large number of cache lines categorised as useful will produce a low WCET, but will increase considerably the cache-related preemption delay. It seems possible to obtain algorithms that modify the vertex categorisation, penalising WCET, but improving the cache-related preemption delay. The authors believe that the focus may be changed: obtaining a good trade-off between the intrinsic and extrinsic cache behavior may provide better schedulability than looking for more accurate cached WCET.

6. REFERENCES

Basumallick, S. and K. D. Nilsen. Cache Issues in Real-Time Systems *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

Busquets-Mataix, J.V, A. J. Wellings, J.J. Serrano, R. Ors and P. Gil. Adding Instruction Cache Effect to an Exact Schedulability Analysis of Preemptive Real-Time Systems. *8th Euromicro Workshop on Real-Time Systems*, 8-15, L'Aquila, Italy, June 1996.

Hennessy, J. L. and D. Patterson. *Computer Arquitecture. A Quantitative Approach* Second Edition. Morgan Kaufmann. San Francisco, 1996.

Kastner, D. and S. Thesing. Cache Aware Pre-Runtime Scheduling *The Journal of Real-Time Systems*, **17**, 235-236.

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

Lee, S., S. L. Min, C. S. Kim, C. Lee, M. Lee Cache-Conscious Limited Preemptive Scheduling *The Journal of Real-Time Systems*, **17**, 257-282.

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.

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.

Mueller, F. and J. Wegener. A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints *Proc. of 4th IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, USA, 1998.

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