Proceedings of the IASTED International Symposia Applied Informatics.
pp. 271-276. Acta Press. Innsbruck, Austria February 2001

USING GENETIC ALGORITHMS IN CONTENT SELECTION FOR LOCKING-CACHES

A. MARTÍ CAMPOY, A. PEREZ JIMENEZ, A. PERLES IVARS, J.V. BUSQUETS MATAIX
Department of Computer Engineering. Technical University of Valencia. SPAIN
{amarti, aperez, aperles, vbusque}@disca.upv.es

ABSTRACT

Modern processors include in their cache memories the ability to preload and lock a set of instructions, avoiding its replacement from cache. This ability may be useful in real-time, multitask systems, where response time of tasks must be a priori known. Locking cache contents makes the system predictable, simplifying the system analysis when calculating execution and response time of tasks. As cache memories improve system performance, real-time systems must take advantage of the speedup given by these memories. Selection of instructions to be loaded and locked in cache must be carefully accomplished to obtain the best performance in addition to predictability.

However, the estimations of the cost to load and lock one instruction over the system response time is not easy due to several factors. Direct search algorithm is ill advised. Also, the size of the problem precludes the use of exhaustive or random search. This paper presents the use of genetic algorithm and its implementation to solve the problem of selecting instructions to be loaded and locked. Experimental results show that implemented algorithm makes a well-directed search. Finally, partial experiments show the usefulness of the presented cache-scheme in real-time, multitask preemptive systems

KEYWORDS

Genetic algorithms, real-time, multitask, locking cache memories.

INTRODUCTION

Cache memories clearly help to improve computer performance. But its use in specific systems, like real-time systems, presents several problems due to its unpredictable behaviour. In real-time, multitask preemptive systems, response time of tasks must be a priori calculated in order to guarantee that all the tasks finish execution before their deadlines. When instruction cache memory is present, execution time of instructions may change over executions, due to the dynamic behaviour of cache. Estimating the response time of a cached task presents two faces: first, calculating the Worst Case Execution Time (WCET), because a task removes its own instructions from cache, making no constant the execution time of each instruction. This effect is called intrinsic interference. Second, calculating the response time of each task, because in a preemptive system, tasks remove from cache other task's instructions, increasing the execution time of preempted task with regard to the execution time without preemptions. This effect is called extrinsic interference.

Several solutions have been presented to solve the problem of using cache memories in real-time systems. [1,2,3] present analytical methods to estimate the WCET of a cached task. [4,5] present analytical methods to estimate the response time of preemptive, multitask cached systems. [6,7,8] present both hardware and software solutions to eliminate the extrinsic interference. But all the proposed solutions solve partly the problem: the intrinsic interference or the extrinsic interference.

Modern processor offers the ability to preload and lock the cache contents, which remain unchanged during system operation. Since no new allocations in cache are possible, both intrinsic and extrinsic interference are eliminated. This allows the system designer to estimate the response time of tasks in a simple way, using well known analysis techniques, since execution time of instructions is constant over executions, and preemptions modify only the task response time, but not increase the execution time by means of extrinsic interference.

However, cache improvement is due to its adaptive behaviour. Locking instructions in cache makes the system deterministic, but in order to obtain the best performance, the preloaded and locked instructions in cache must be carefully selected. The number of possible solution for a specific combination of a set of tasks and cache size may be very large, so an efficient algorithm is necessary. This paper presents a solution to this problem, using a genetic algorithm [9] to find with low temporal cost a sub-optimal set of instructions to lock in cache. The experimental results show that the algorithm fits this problem, finding good solutions with a low temporal cost. Also, partial experiments show the usefulness for real-time systems of cache schema here presented.

PROBLEM DESCRIPTION

The cache scheme presented in this paper is a full-associative instruction cache. This cache can be preloaded and locked in a per block basis. A block is the minimum unit of information that can be transferred from main memory to cache. In this paper, size of block, also called cache line, is sixteen bytes. Once the cache is preloaded with the desired blocks, its content is locked, precluding new loads in cache and the replacement of already loaded blocks.

When the processor references an instruction belonging a preloaded block, the instruction is served by cache memory, like a cache hit in non-locked cache. If the reference miss in cache, the entire block is transferred from main memory to a buffer of one cache-line size. Consecutive references to this block hit in the buffer, and are served like a cache hit, until a new, non-preloaded block is transferred to the buffer. With this cache schema, execution time of each instruction is constant and well known, therefore timing analysis of the entire system is very easy.

Due to the different size of cache and tasks not all the instructions can be loaded and locked in cache. Randomly selecting instructions to load is enough to make the system deterministic, but may reduce the system performance, since loaded instructions execute faster than the non-locked instructions. This way, an efficient algorithm is needed to select a set of instructions to preload in cache, obtaining the best possible performance when using locking caches.

An exhaustive search, including branch and bound is not possible due to the size of the problem. For example, for a set of task with 5.000 instructions executing in a cache of 2KB, represents more than 10^{50} different solutions. This value invalidates any chance to exhaustive or random search.

Since the response time of tasks depends on the response time of higher priority tasks, it is not easy to isolate an instruction to estimate the effect of locking it in cache over the system response. Thus, a direct algorithm to find the best set of instructions to load and lock may have an intractable complexity, both in implementation and temporal cost.

Genetic algorithm, which combines direct and random search, fits well in this kind of problems. The next section describes in detail the use of these algorithms to find a sub-optimal solution to the presented problem.

DESCRIPTION OF THE GENETIC ALGORITHM

The use of genetic algorithm in any search problem requires the following elements and operators:

- A representation of the solutions, called codification.

- A function to evaluate the different solutions, called fitness function.

- A selection-scheme in order to choose a subset of solutions in order to create new solutions.

- Crossover and mutation operations used to transform selected solutions in new solutions.

- Initialisation of the first set of solutions, and tuning the parameters that govern the above operators.

Codification

The genetic algorithm works over a set of possible solutions, each one called individual. The set of individuals is called population. Let n be the number of memory blocks occupied by all the tasks. The individual is a binary vector of dimension n. If a bit is 1, the related memory block is preloaded and locked in cache. If a bit is 0, it means that the related block is not loaded in cache.

Fitness function

The fitness function evaluates the goodness of each individual. The fitness function may be the same that the target function to optimise, but usually another function is used instead of the target function, in order to improve the response time or the way the algorithm explore the search space. In the presented algorithm, the fitness function is the weighted average of all tasks response time, while the target function is the average of all tasks response time.

The response time of each task is calculated using the Response Time Analysis. Since the cache contents are unchanged during preemptions, no extensions are needed to the RTA. The RTA needs the Worst Case Execution Time of each task. The WCET of each task is calculated using a single path analysis [10]. In this analysis, cache effect is considered in the execution time of each instruction in the following way:

- If an instruction belongs to a memory block loaded and locked in cache, its execution time is *Thit*, time of execution from cache.

- If an instruction belongs to a memory block not loaded in cache and it is the first instruction in a block that the program executes, its execution time is *Tmiss*, time of execution from main memory.

- If an instruction belongs to a memory block not loaded in cache and it is not the first instruction in a block the program executes, its execution time is *Thit*, because this instruction is loaded in the temporal buffer.

The equation 1 represents the calculation of fitness function, where R_i is the response time of task T_i, and T_i has a higher priority than task T_j if $i < j$.

$$\text{Fitness} = \frac{R_1 + R_2 + 2R_3 + 4R_4 + .. + 2^{n-2} R_n}{2^{n-1}} \quad [1]$$

The use of this fitness function is based on the relationship between tasks. In a preemptive system, response time of task T_i depends of its execution time and the response time of higher priority tasks. Let consider three tasks: T1, T2 and T3. T1 has the highest priority and T3 the lowest. Locking in cache instructions of T3 reduces its execution time and its response time, but do not affect to T1 and T2. However, locking instructions of T1 reduces its execution time and its response time, but also reduces response time of T2 and T3. This way, to avoid that the entire cache will be assigned to the highest priority task, the fitness function is weighted towards lower priority tasks.

Selection-scheme

From the fitness function four types of results are obtained:

- Finite average value, with number of locked blocks less or equal to the cache size. This is a valid individual.

- 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 less or equal to the cache size. Due to the large execution time of tasks, some tasks never finishes its execution. 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.

These four possibilities preclude the use of a fitness-proportionate selection schema. Instead of it, rank-based selection is used, where the probability of select one individual is function of its position, and not of its fitness value [11]. The individuals are arranged considering both the fitness value and the number of locked blocks. Higher positions are assigned to valid individuals, using the fitness value (also for infinite values, where WCET is used instead fitness to arrange two individuals with infinite fitness values). The lowest positions are assigned to non-valid individuals, arranged as function of the number of locked blocks.

Crossover and mutation

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

Crossover usually produces new individuals with number of locked blocks greater than cache size. In order to raise the probability of getting valid individuals in the population, mutation is applied in three ways:

For individuals with number of locked blocks greater than cache size, mutation randomly selects a set of locked blocks and mark them as unlocked, reducing the number of locked blocks. The resulting individual may have a number of locked blocks that are greater, equal or lower than cache size.

For individuals with number of locked blocks lower than cache size, mutation randomly selects a set of unlocked blocks and mark them as locked, increasing the number of locked blocks. The resulting individual may have a number of locked blocks greater, equal or lower than cache size.

For individuals with number of locked blocks equal than cache size, mutation randomly selects a set of pairs, each pair with one locked block and one unlocked block, and exchange them, leaving unchanged the number of locked blocks.

Initial population and tuning parameters

Although a genetic algorithm can explore all the search space through crossover and mutation, selecting adequately the initial population may help the algorithm to find a sub-optimal solution with a minor number of iterations. In the presented problem, the best solution is an individual with a number of 1's equal to the cache size. Due to the structure of tasks the best solution includes a large sequence of consecutive 1's. The population is initialised with sequences of 1's, randomly selecting the beginning.

Other parameter settings are:

- Population size: 200

- Number of generations: 2000

- Probability of crossover: 0.6

- Probability of mutation for individual with number of locked blocks equal to cache size: 0.01

- Probability of mutation for individual with number of locked blocks distinct to cache size: 0.001

- Probability of selection of the highest ranked individual: 0.1

The parameter settings are based on results of several preliminary runs. They are comparable to the typical values mentioned in the literature [11].

EXPERIMENTAL RESULTS

This section reports the results of four experiments that show that the genetic algorithm presented can find a sub-optimal solution. The algorithm has been implemented in C and executed in a medium range personal computer, using Linux as operating System. Execution time of algorithm depends on problem data, because the RTA is an iterative algorithm. The number of iterations of RTA depends on value of the WCET of each task, and this value depends on the size of cache used for the experiment. However, the execution time of genetic algorithm never exceeds ten minutes.

The first experiment is formed by five tasks of similar size. Only the lowest priority task is formed mainly by a loop of 1000 iterations. This set of simple tasks allows verifying the correctness of the algorithm. Table 1 shows how the algorithm distributes the cache along the tasks, loading and locking those memory blocks that reduce the average execution time. The columns indicate the number of cache lines assigned to each task.

Cache size	Task 1	Task 2	Task 3	Task 4	Task 5
0.5 KB	0	0	0	0	32
1 KB	0	0	0	0	64
2 KB	0	0	0	0	128
4 KB	112	2	1	0	141
8 KB	175	169	27	1	141
16 KB	175	169	156	155	144

Table 1. Cache assignment for experiment 1.

For the set of tasks of experiment 1, only instructions in task 5 profit from cache, because only these instructions execute more than one time. While the cache size is smaller than task 5 (2KB) the algorithm select instructions from this task. When the cache size is greater than task 5 size, the algorithm selects instructions from the highest priority tasks, following the selection in order of priority. This is due to preemptions, so reducing the execution time of high priority tasks reduces the response time of low priority tasks. Notice that for task 5, a set of instructions outside the loop is selected only when all the tasks fit in cache.

Second experiment uses the same set of tasks than experiment 1, but the fourth task is formed mainly by a loop of 500 iterations. Task 5 remains unchanged from experiment 1. Table 2 shows how the algorithm distributes the cache along the tasks. In this experiment, while the cache size is smaller than tasks size, the algorithm selects instructions from task 4. Instructions in task 4 benefit from cache because they are inside a loop, and reducing its execution time will also reduce the response time of task 5. But when cache size is greater than the size of task 4, the algorithm selects instructions from task 5, and not from task 1 like in experiment 1, because instructions of task 5 are inside a loop.

Cache size	Task 1	Task 2	Task 3	Task 4	Task 5
0.5 KB	0	0	0	32	0
1 KB	0	0	0	64	0
2 KB	0	0	0	128	0
4 KB	0	0	0	147	109
8 KB	175	47	2	147	141
16 KB	175	169	156	155	144

Table 2. Cache assignment for experiment 2.

Third experiment uses the same tasks than previous experiments but each task is formed by a loop of 10 iterations. Table 3 shows the cache assignment for each task. In this case, the algorithm selects first instructions from the highest priority task, and follows in priority order.

Cache size	Task 1	Task 2	Task 3	Task 4	Task 5
0.5 KB	32	0	0	0	0
1 KB	63	1	0	0	0
2 KB	127	1	0	0	0
4 KB	172	83	1	0	0
8 KB	172	165	152	23	0
16 KB	172	169	156	151	144

Table 3. Cache assignment for experiment 3.

Fourth experiment is formed of three tasks. Tasks 1, the highest priority task has two nested loops with a total size about 3000 instructions. Task 2 is a single loop with a if-then-else structure, with total size about 6000 instruction. Task 3, the lowest priority task is a single loop with near 8000 instructions. Figure 1 shows the evolution of the fitness function of the best individual for the 2000 iterations with a cache size of 16KB. For each iteration the best individual presents a lower value than the best individual of previous iteration. Only in some cases, more frequently for last iterations, the new individual presents no improvement. This is due to the fast convergence of

the algorithm, finding the best solution in the first iterations.

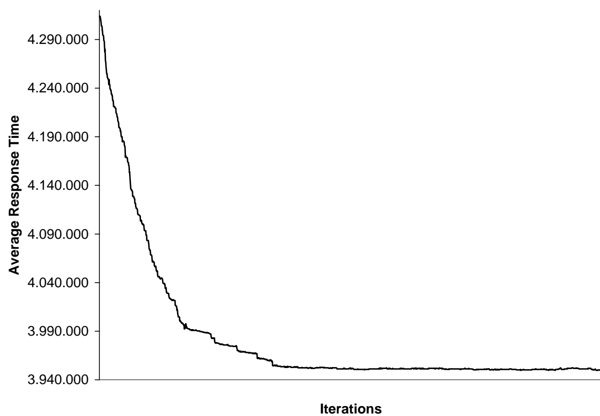

Figure 1. Evolution of fitness function with 16KB of cache.

Figure 2 shows the average response time for the set of tasks of experiment 2 using several cache sizes. The labels "direct", "2-set", "4-set" and "full" represent the average response time for, respectively, a direct-mapped cache, a two-set associative cache, a four-set associative cache and a full associative cache. The label "estimated" represents the average response obtained from the genetic algorithm, and "simulated" shows the average response time using the locking cache, loading and locking the set of memory blocks selected by the genetic algorithm. The simulation has been accomplished using an extended version of the SPIM simulator [12].

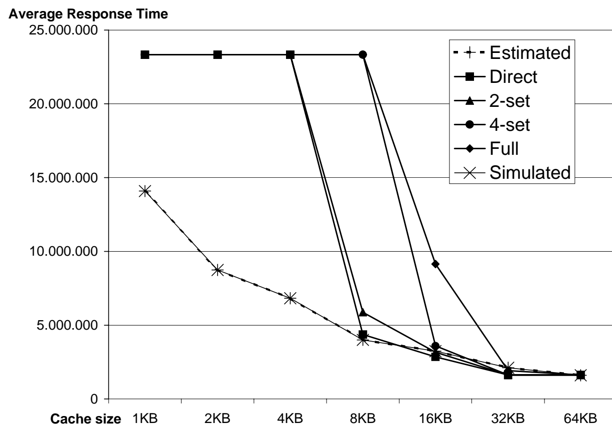

Figure 2. Average Response Time for experiment 2.

Figure 2 shows, firstly, that values given by the genetic algorithm are an upperbound of the actual response time using a locking cache (numerical values resulting from the simulation of the locking cache are 99'9 % related to estimated values). Secondly, the response time of locking cache is, in the worst cache, lightly greater than the response time using conventional caches, but in several cases the performance of the locking cache is better. These results validate both the genetic algorithm and the cache scheme, providing excellent solutions. However, regarding the performance of locking cache, more experiments must be accomplished, but this is out of the scope of this paper.

CONCLUSIONS

This paper presents a genetic algorithm to select the instruction to be loaded and locked in a locking cache. The set of instructions selected provides the best possible performance.

The algorithm described in this paper efficiently solves the problem presented for the use of locking caches. Three are the major advantages of the algorithm: first, the low computational cost to find a sub-optimal solution. Second, the algorithm well fits to parameters of the problem, as structure of tasks or cache size. Third, the algorithm gives both the set of memory blocks to load and lock and an upperbound of the system response time. Thus, system designer obtains simultaneously the set of instructions to be loaded and the schedulability analysis.

The predictability given by the locking cache, together with the algorithm, makes its use very interesting on preemptive real-time systems, in contrast with the use of a conventional cache. In the latter case, it is mandatory a complex analysis to obtain an accurate estimation of the response time of cached tasks.

At the moment, the algorithm neither uses information about task structure nor problem parameters. Using a more complex representation of the problem, including, for example, number of iterations of each loop, could help the algorithm to find the solution faster. Also, the algorithm used to calculate the execution time of tasks, a single path analysis, may be improved to get a more accurate value.

The cache-scheme introduced in this paper may also improve system performance, in addition to ensure the predictability. Thus, response time of tasks is significantly reduced for some cache sizes.

ACKNOWLEDGMENTS

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

REFERENCES

[1] F. Mueller 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*, 1998.

[2] S. S. Lim, 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*, 1994.

[3] Y. S. Li, 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*, 1996.

[4] J. V. Busquets-Mataix, 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*, 1996, 8-15.

[5] C. Lee, J. Hahn, Y. 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. *Proc. of the 18th IEEE Real-Time Systems Symposium*, 1997.

[6] D. B. Kirk. SMART (Strategic Memory Allocation for Real-Time Cache Design. *Proceedings of the 10th IEEE Systems Symposium*, pages, 1989, 229-237.

[7] J. V. Busquets Mataix, J.J. Serrano, A.J. Wellings. Hybrid Instruction Cache Partitioning for Preemptive Real-Time Systems. *9th Euromicro Workshop on Real-Time Systems*, 1997 , 271-276.

[8] A. Wolfe. Software-Based Cache Partitioning for Real-Time Applications. *Proceedings of the 3th International Workshop on Responsive Computer Systems*, 1993.

[9] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley Co., Inc, 1989.

[10] A. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Softwae Engineering,* 15(7), 1989, 875-889.

[11]M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

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