A Comparison of Three Genetic Algorithms for Locking-Cache Contents Selection in Real-Time Systems

E. Tamura[1], J.V. Busquets-Mataix[2], J. J. Serrano Martín[2], A. Martí Campoy[2]
[1]Grupo de Automática y Robótica, Pontificia Universidad Javeriana - Cali, Colombia.
[2]Departamento de Informática de Sistemas y Computadores*,
Universidad Politécnica de Valencia, Spain
E-mail: eutamo@doctor.upv.es, {vbusque, jserrano, amarti}@disca.upv.es

Abstract

Locking caches, providing full determinism and good performance, are a very interesting solution to replacing conventional caches in real-time systems. In such systems, temporal correctness must be guaranteed. The use of predictable components, like locking caches, helps the system designer to determine if all the tasks will meet its deadlines. However, when locking caches are used in a static manner, the system performance depends on the instructions loaded and locked in cache. The selection of these instructions may be accomplished through a genetic algorithm.

This paper shows the impact of the fitness function in the final performance provided by the real-time system. Three fitness functions have been evaluated, showing differences in the utilisation and performance obtained.

1 Introduction

To cope with the processing power demanded by today complex engineering systems, real-time system designers are resorting to high-performance contemporary processors. These processors are designed with a major goal: to provide good average execution times over a great variety of computing needs. Unfortunately, in real-time systems, what the designer needs is to guarantee that the tasks must execute on time even under adverse circumstances. To do so, it is necessary to estimate the worst-case response time of every task in the system.

Given the speed disparity between the memory system and contemporary processors, computer architects introduce a cache memory in between. However, cache operation makes hard to predict the execution times since cache contents change dynamically, adapting to the execution path in each moment. Furthermore, during its execution, the tasks working sets interferes with each other. Therefore, to ease the analysis, it is feasible to lock the contents of the cache memory with some predetermined instructions such that the system provides not just better performance but also predictability as shown in ref. 1 and ref. 2. The problem is that, as it is proved in ref. 3, determining an optimal placement of cache contents by trying to maximise the number of times that the referenced datum is in the cache memory is NP-hard. Hence, it may be a good idea to apply some heuristics. Previous works published in ref. 4 show that using a genetic algorithm represents a good solution for this particular problem.

However, performance of locking cache is highly dependant on the genetic algorithm characteristics. In addition, performance of real-time systems must be quantified with several metrics: system utilisation, task slack (distance from end of task execution to its deadline), or average task slacks in the system.

This paper explores three different fitness functions, and evaluates the performance they provide in terms of system utilisation and slack time, two metrics commonly used in evaluating real-time systems. The results will illustrate which is the most adequate fitness function to get the best value for each metric.

The remainder of the paper is organised as follows. Section 2 introduces the genetic algorithm and its three fitness functions. Section 3 illustrates the experimental procedure. Finally, in Section 4 some conclusions and future work are summarised.

2 Genetic Algorithm

2.1 Representation

Each individual is modelled as a binary vector of dimension n, where n is the number of memory blocks occupied by all of the tasks. If a bit is set to one, the associated memory block is to be loaded and locked in the cache memory.

2.2 Fitness functions

The first fitness function will attempt to allocate more blocks for those real-time tasks with lower priorities, try-

*This work has been supported in part by the Spanish *Comisión Interministerial de Ciencia y Tecnología* under project CICYT-TIC2003-08106-C02-01

ing to compensate the time lost due to the execution of higher priority real-time tasks. Equation 1 shows the resulting fitness function:

$$f_A = \frac{R_1 + \sum_{i=2}^{N} 2^{i-2} R_i}{2^{N-1}} \tag{1}$$

where N is the number of tasks in the system, R_i is the response time of each task, τ_i, computed using CRTA as given in ref. 5.

The second fitness function will try to provide a lower processor utilisation at the system level.

$$f_B = \sum_{i=1}^{N} \frac{C'_i}{T_i} \tag{2}$$

where C'_i, the computation time of τ_i includes all cache effects.

The third fitness function tries to improve the average slack time at the system level.

$$f_C = \frac{\sum_{i=1}^{N} S_i}{N} \tag{3}$$

with

$$S_i = 1 - \frac{R_i}{D_i} \tag{4}$$

where R_i and D_i are respectively the response time and deadline of τ_i.

2.3 Selection Criteria

Given any of the previous fitness function there are two possible outcomes:

- The number of locked blocks required by the task set is less than or equal to the cache memory size. This is a valid individual.
- The number of locked blocks required by the task set is greater than the cache memory size. This is a non-valid individual, since even though the system may provide acceptable response times, it does not satisfy the intended requirements.

A rank-based selection is used, in which the individuals are sorted according to two basic rules. If the individual is valid, it is ranked according to its fitness value. If the individual is not valid, it is ranked according to the number of blocks used; the lesser the number of blocks, the better the individual.

2.4 Crossover and Mutation

For crossover, two individuals are randomly selected from the previous ranking. Both individuals are divided in two ends, randomly selecting the splitting point. Then, by exchanging the two portions to the right of the cut-off point, two new individuals are created. Once the crossover is done, the resulting new individuals may use more cache lines than the cache memory has available. However, to make a broader exploration of the search space, those individuals are not discarded. Therefore, mutation is applied by following one of three schemas:

- For individuals with a number of locked blocks greater than cache size, the mutation procedure selects at random a set of locked blocks and marks 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 the cache size.
- For individuals with a 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 the cache size.
- For individuals with a 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 exchanges them, leaving unchanged the number of locked blocks.

This policy allows the existence of non-valid individual, but also helps to keep its number low.

2.5 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 lower number of iterations. Due to the structure of the tasks, the ideal solution is an individual with a number of 1's equal to the cache size; hence, 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: 5000; 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 ref. 6.

3 Experimental Procedure

There were 26 different setups, each with 3 to 8 tasks. The code for each task is synthetic; it does nothing useful but it has a mix of instructions such that it is easy to generate different programs, which is adequate for the purpose. Each experiment was tested using seven different cache memories ranging from 64 lines to 4096 lines, for 182 experiments comprising 770 tasks. In some of the experiments, the footprint (the amount of memory required) of the task set was smaller than the cache size, which means that they will run as fast as possible, since there will be no interference at all. Because of this, they were discarded; the final number of valid experiments is 146 and the number of tasks is 610.

For each of the 146 experiments, three runs of the genetic algorithm -one run for each fitness function- were accomplished. For each run, the overall system utilisation, the per-task slack times, and, finally, the overall slack time, were estimated.

3.1 Overall System Utilisation

Let N be the number of tasks in the system, the computation time, C'_i, of each task, τ_i, is calculated using CRTA. Assuming that task τ_i has higher priority than task τ_j whenever $i < j$, the calculation of the computation times, are given by:

$$C'_1 = R_1 \quad (5)$$

$$C'_i = R_i - \sum_{\forall\, 0<j<i} C'_j \left\lceil \frac{R_i}{T_j} \right\rceil, \forall\, i \mid 1 < i \leq N (6)$$

which takes into account the execution times of those tasks whose priority is higher than the priority of the current task.

The system utilisation, U_l, is then given by:

$$U_l = \sum_{i=1}^{N} \frac{C'_i}{T_i} \quad (7)$$

3.2 Slack Time

Let N be the number of tasks in the system, the per task slack time, S_i, is given by:

$$S_i = 1 - \frac{R_i}{D_i} \quad (8)$$

where D_i is the deadline of task τ_i.

Then, the system average slack time is calculated by means of:

$$S_{avg} = \frac{1}{N} \sum_{i=1}^{N} S_i \quad (9)$$

3.3 Statistical Analysis

With just the bare results attained in the experiments it is not possible to declare, which fitness function is better, so statistical analyses will be used. First, a comparison of the statistical summaries is done; this is followed by a paired-sample analysis, a procedure designed to test for significant differences between two data samples where the data is collected as pairs. Several statistical values have been used to determine if there exist differences between samples. In addition, three null hypothesis tests have been done for each comparison. These are t-test, sign test, and signed rank test.

3.3.1 Analysis of System Utilisation: Figure 1 shows a comparison of the average, median, lower and upper quartile for the utilisation obtained from each fitness function. It can be observed that function f_A provides the largest (the worst) utilisation, while functions f_B and f_C offers similar results. In order to determine if there are significant differences between the utilisation provided by f_B and f_C, a paired-sample analysis has been performed. The analysis shows that the difference between f_B and f_C is small but statistically significant, giving lower utilisation (better performance) for the results obtained when using the fitness function f_B.

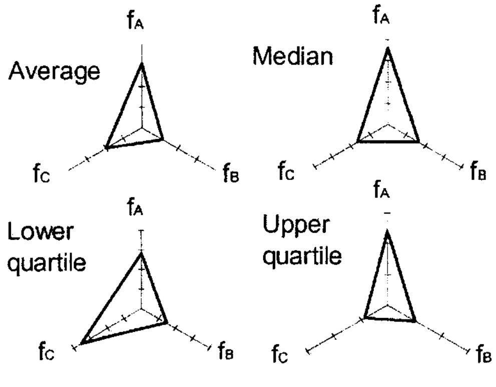

Fig. 1. Utilisation from each fitness function

3.3.2 Overall Slack Time: From Figure 2 it can be clearly recognised than f_C offers a large (better) average slack than f_A and f_B. Regarding the comparison between f_A and f_B, Figure 2 presents contradictory results. Besides that, the paired-sample analysis for the data coming from f_A and f_B does not help to decide which of the two fitness functions provides better average slack. Albeit, it may be reasonable to say that both f_A and f_B provide the same average slack.

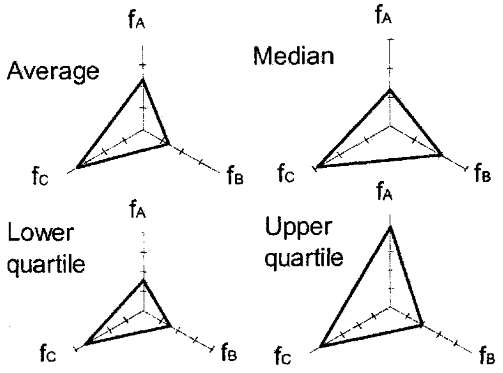

Fig. 2. Average slack from each fitness function

4 Conclusions and Future Work

This paper showed the impact of three different fitness functions over the results provided by a genetic algorithm that selects the contents of a locking cache memory, which is used in a static way in a real-time system. The first fitness function, f_A, tries to minimise the average response time of the tasks. The second one, f_B, tries to minimise the global system utilisation. Finally, the third one, f_C, tries to minimise the average slack time.

The evaluation is based upon two metrics: global system utilisation, U, and Overall Average Slack Time, S.

- Fitness function f_B is statistically better than f_C whenever it is required to optimise the overall utilisation. Fitness function f_A presents worst utilisation than the other two functions.
- Regarding overall slack time, fitness function f_C offers the better performance, while there is no difference between f_B and f_A.

From the results it can be seen that none of the fitness functions perform well in the two basic metrics, although fitness function f_C may be the best option to get the optimal average slack and quasi-optimal utilisation. However, if the performance required for one of the two metrics is a critical parameter, the system designer should choose between f_B and f_C the one that it is more appropriate to its particular optimisation interests. Table 1 gives the proposed selection criteria of fitness function.

Table 1. Function selection criteria versus optimising parameter

Rank	Optimising for utilisation	Optimising for slack
First option	f_B	f_C
Second option	f_C	f_B or f_A
First option	f_A	f_B or f_A

The statistically significant but very small difference between f_B and f_C concerning utilisation allows to expect that the combined use of the two fitness functions in the same genetic algorithm, may bring a trade-off solution for both metrics. Therefore, future work will involve the development of a selection operator that considers two or more metrics for different fitness functions in order to sort the individuals.

References

[1] Martí, A., Perles, A., Busquets Mataix, J. V. (2001) Static Use of Locking Caches in Multitask Preemptive Real-Time Systems. In: Proceedings of the IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the 22nd IEEE Real-Time Systems Symposium), London, UK.

[2] Martí, A., Perles, A., Busquets-Mataix, J.V. (2002) Dynamic Use Of Locking Caches In Multitask, Preemptive Real-Time Systems. In: Proceedings of the 15th World Congress of the International Federation of Automatic Control, Elsevier Science, Barcelona, Spain.

[3] Petrank, E., Rawitz, D. (2002) The harness of cache conscious data placement. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 101-102, Portland, U.S.A.

[4] Martí Campoy, A., Pérez Jiménez, A., Perles Ivars, A., Busquets Mataix, J.V. (2001) Using Genetic Algorithms in Content Selection for Locking-Caches. In: Proceedings of the IASTED International Symposia Applied Informatics. pp. 271-276. Acta Press. Innsbruck, Austria.

[5] Busquets-Mataix, J.V., Wellings, A.J., Serrano, J.J., Ors, R., Gil, P. (1996) Adding Instruction Cache Effect to an Exact Schedulability Analysis of Preemptive Real-Time Systems. In: Proceedings of the 8th Euromicro Workshop on Real-Time Systems, pp. 8-15, L'Aquila, Italy.

[6] Mitchell, M. (1996) An Introduction to Genetic Algorithms, MIT Press, Cambridge