An Algorithm for Deciding Minimal Cache Sizes in Real-Time Systems

Antonio Martí Campoy
Computer Engineering Department
Universitat Politècnica de València
46022 València, Spain
amarti@disca.upv.es

Francisco Rodríguez-Ballester
Computer Engineering Department
Universitat Politècnica de València
València, Spain
prodrig@disca.upv.es

Eugenio Tamura
Grupo de Automática y Robótica
Pontificia Universidad Javeriana - Cali
Calle 18 118-250
Cali, Colombia
tek@javerianacali.edu.co

Rafael Ors
Computer Engineering Department
Universitat Politècnica de València
València, Spain
rors@disca.upv.es

ABSTRACT

When designing real-time systems, predictability is of utmost importance. A locking cache is a cache memory that allows loading and locking instructions, thus avoiding their replacement. This way, regarding memory accesses, execution time of instructions is constant since it does not depend on the sequence of memory references. With a predictable behaviour, locking cache memories are a practical alternative to conventional caches for real-time systems. Offering similar performance to conventional caches, locking caches allow an accurate yet simple schedulability analysis.

Locking caches may also help to reduce the size of a system, by means of reducing cache size. When reducing cache size, also cost and power consumption may be reduced. This way, both predictability and cost saving is provided by means of locking cache.

This work presents a set of algorithms, aimed to select the contents of a locking cache that provides the minimum locking cache size, while the system remains schedulable. Compared to a previous approach, the algorithms presented in this paper are able to select a set of main memory blocks that result in a smaller cache size

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; B.3.2 [**Memory Structures**]: Performance Analysis and Design Aids—*Worst-case analysis*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods, and Search - Heuristic methods*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'11, July 12–16, 2011, Dublin, Ireland.
Copyright 2011 ACM 978-1-4503-0557-0/11/07 ...$10.00.

General Terms

Algorithms, Performance, Design.

Keywords

Real-Time Systems, Locking-Cache Memory, Schedulability, Genetic Algorithms, Greedy Algorithms.

1. INTRODUCTION

Locking cache memories have been proposed as an alternative to conventional cache memories in order to obtain predictable behaviour and higher performance in real-time systems [5] [6] [7] [11] [17]. The major drawback of a conventional cache memory lies in its intrinsic dynamic and adaptive behaviour, which makes estimating tasks execution and response times complex.

When a locking cache memory is used, a set of blocks of main memory containing instructions is selected, loaded in cache memory and then locked. Since the contents of the cache memory is known, it is possible to apply simple, well-known techniques and methods to estimate execution and response times: [14] to calculate the tasks Worst Case Execution Time (WCET) and [1] for computing the tasks response time and analysing the system schedulability taking into account the Cache Refill Penalty, CRTA (Cache Response Time Analysis).

The major goal of using a locking cache is thus to ease the estimation of execution and response times and the schedulability analysis. Furthermore, in some cases, it makes this analysis feasible.

Nonetheless, locking caches can also be useful for other purposes. In [4] a new application of locking cache memories in the area of real-time systems is presented. It targets application-specific optimised systems and the purpose is to reduce the cache memory size to a minimum, thus diminishing the total system cost by lowering the hardware cost and allowing integrating the cache in a System-on-a-Chip. The result: energy savings and increased computing efficiency while satisfying real-time requirements.

The approach used is termed Static Locking and works as follows [5]: before the system starts executing, a locking instruction cache memory is loaded with the selected instructions; the contents remain without modifications nor replacements while the system is operating. The loaded and then locked instructions may belong to any task in the system. That is, the available cache lines can be shared by all of the tasks in the system.

The main benefit from using a locking cache is to get a predictable system regarding memory accesses, which allows to use simple techniques to perform the schedulability analysis. On the other hand, the performance offered by the system should be at least close to that offered when using a conventional cache memory. Thus, in order to improve the resulting performance, the instructions to be loaded and locked into cache memory must be chosen very carefully since this selection directly affects the execution and response times of the system tasks.

This problem has been solved with two different kinds of algorithms. The first one [2] is a genetic algorithm [9] whose goal is to minimise the system utilisation. In the second algorithm, to minimise execution times, a greedy algorithm [13] is used; in it, the selected instructions are those which are executed more frequently.

The output of both algorithms is a subset of the instructions in every system task; those instructions constitute the set of instructions that must be loaded and then locked into cache memory. In [3] it is shown that the solution quality, measured as the performance obtained when locking the chosen set in a cache memory, is very similar for both algorithms; in some cases however, the genetic algorithm offers better results than its counterpart. Nonetheless, the execution time of the genetic algorithm is higher than that of the greedy one.

2. THE PROBLEM

In [4] the genetic algorithm, as described in [2], was used to select the contents of the locking cache. This algorithm has as one of its input parameters the size of the locking cache memory. However, the problem now is to determine the size of the locking cache. Thus, the genetic algorithm was executed as many times as necessary, using different cache sizes, to find the minimum size that keeps the system schedulable.

Although this approach shed light on both its feasibility and suitability, it exhibits two main drawbacks: The first one is the temporal cost. Even when using a dicotomic search, the number of executions can be very high. In the reported experiments [4], the number of executions of the genetic algorithm took into account line sizes between 1 and 4096, which leads to 12 executions in the worst case.

The second problem is that the goal of the genetic algorithm was to obtain the best possible performance; that is, to make the system schedulable minimising the system utilisation at the same time. However, as the cache memory size diminishes, the system utilisation increases, which may cause individuals to become unschedulable (i.e., utilisation is higher than 100%). When performing selection and crossover operations, non-schedulable individuals with smaller cache sizes have a low probability of being chosen. Even so, as the cache size approaches the minimum, the algorithm exhibits some problems finding valid (that is, schedulable) individuals.

3. THE PROPOSAL

This paper presents four different algorithms whose goal is to obtain the minimum cache size required for a real-time system to be schedulable. For each algorithm an assessment on both response time and solution quality is done. The algorithms are:

- Prag-Down,
- Prag-Up,
- GA,
- Prag-GA.

A description of each is given in following subsections.

3.1 Prag-Down

This algorithm does not seek for the minimum cache size. Instead, it repeatedly executes a sub-optimal locking cache contents selection algorithm. The algorithm is the pragmatic one as described in [13]. Originally, this algorithm was designed to accept the cache size as a parameter input. Nevertheless, its small execution time (about one minute in most of the cases) allows its integration into a descending sequential search.

The iterations start using the maximum cache size and then each new execution uses one cache line less than the previous one; when the system becomes non-schedulable, the iterations finish.

3.2 Prag-Up

Prag-up uses the same algorithm and approach described in Prag-down. This time however, the search goes up instead of going down: the first iteration starts with one cache line, and for every new execution one more line is added. The iterations stop when the system becomes schedulable.

The rationale behind this approach is to make sure that there are no discontinuities on the cache size that make the system schedulable. That is, assuming that a given system using N cache lines is schedulable but non-schedulable when using $N-1$ cache lines, it is necessary to ascertain whether the system is non-schedulable for any size lower than $N-1$.

3.3 GA

This is the sequential version of the genetic algorithm presented in [15]. Eight different versions have been evaluated. All of them have in common the problem representation, the fitness function and the crossover operation; they differ in the procedure used to create the initial population, and both the selection and mutation operations. In the following the operators and different variants are described:

3.3.1 Problem Representation

A tri-dimensional matrix stores the status of main memory blocks in order to determine whether they are selected or not to be loaded into the locking cache. The first index identifies to which task the memory block belongs. The second index identifies the set or cache memory line in which the block is mapped. The third dimension is used to store the list of blocks that are mapped to that set together with their corresponding status.

This kind of structure, rather complex in appearance, allows crossing two individuals and mutating the resulting individuals guaranteeing that they will not use more blocks than the available ones. Furthermore, the cache mapping function [10] will not be violated; as a way of example, assuming that the mapping function is direct, once the crossover and mutation operations are done, it is guaranteed that any two different blocks that map to the same cache line will not be selected simultaneously.

3.3.2 Initial Population

Two different ways of creating the initial population will be evaluated. The first one, termed *i1*, creates identical individuals; each individual has all of the blocks for every task marked as selected for being loaded and locked in cache memory. From this population, as the GA evolves it eliminates selected blocks to reduce cache size.

The second initialisation policy, termed *i2*, creates a single individual with all the blocks of all tasks marked as selected for being loaded and locked in cache memory; the rest of the individuals are created by selecting a pseudo-random number of blocks. The purpose of including the special individual is to broaden the initial search space and ensure the algorithm starts with some schedulable individuals.

3.3.3 Fitness function

The fitness function, which is common for every variant of the GA, results from the combination of the result of the schedulability test, the number of used cache lines, and the system global utilisation. In this work the fitness value for an individual is estimated by means of a 3-tuple (S, L, U), where S is the boolean result from the schedulability test; L is the cache size, in lines, that the individual needs; and U is the global utilisation of the system.

An individual i with fitness (Si, Li, Ui) is better than any individual j with fitness (Sj, Lj, Uj) if: $(Si\ and\ not(Sj))$ or $(Si\ and\ Sj\ and\ (Li < Lj))$ or $(Si\ and\ Sj\ and\ (Li = Lj)\ and\ (Ui < Uj))$. That is, a given individual is better than another if it is schedulable and the other is not, or if it uses a smaller cache (both being schedulable), or in case of tie, the former presents a lower system utilisation.

The schedulability test determines whether all the tasks in the system meet their deadlines. The schedulability analysis is accomplished by comparing the response time of tasks versus their respective deadlines.

In order to compute the response time of the tasks, it is necessary to estimate both their WCET and CRTA. This poses no problem due to the use of a statically-locked instruction cache; these figures can be easily obtained by using the equations described in [5]. Finally, from the response time of tasks, it is easy to compute the system global utilisation.

Although it is possible to use other metrics such as task slack or response times, in [16] it is shown that using the system utilisation in the fitness function offers similar results than using these aforementioned metrics.

3.3.4 Selection policy

To reduce the execution time of the GA, binary tournament has been chosen for the selection policy; to select a parent, two individuals are chosen pseudo-randomly, their fitness functions are compared, and the better individual is choosen. The previous procedure is repeated to choose the other parent.

Besides, an elitist selection with two variants has been introduced. The first variant, termed *s1*, makes two copies of the best individual in the current generation. Since these two copies are subject to mutation, it may be possible that the best individual does not compete in the next generation. The second variant, termed *s2*, also makes two copies of the best individual. In this case however, one of them suffers no mutation and is effectively incorporated into the next generation without any modification.

3.3.5 Crossover operation

Single point crossover is used in this algorithm. The single-point crossover is obtained by splitting every parent at a locus given by a pair $Task\ number$ and $Set\ number$. This approach guarantees that when any two parents are joined, the resulting individual satisfies the cache mapping function.

3.3.6 Mutation operation

This operation also has two variants. In the first one, termed *m1*, one out of three operations is made:

- pseudo-randomly select a set of unlocked blocks and mark them as locked, thus increasing the number of locked blocks, and therefore, the size of cache;
- pseudo-randomly select a set of block pairs, each pair containing a locked block and an unlocked one, and exchange them, leaving unchanged the number of locked blocks and the cache size;
- pseudo-randomly select a set of locked blocks and mark them as unlocked, thus decreasing the number of locked blocks and therefore, the cache size. This mutation schema is derived from the original GA, and the authors judged that it was worth to assess its performance.

The second variant, termed *m2*, pseudo-randomly selects a set of locked blocks and marks them as unlocked, thus decreasing the number of locked blocks and therefore, the cache size. The purpose of this variant is clear: attempt to reduce as soon as possible the cache size.

3.3.7 Termination condition

The GA execution finishes when a predetermined number of generations has been reached. This value and the remaining input parameters will be discussed in the Experiments section.

3.4 Prag-GA

This algorithm results from merging the pragmatic algorithm and the *GAi2s2m2*. The latter was chosen because it is the GA version which offers better results as will be shown in the Experimental Results section. In [3] it is shown that both algorithms offer results of an almost identical quality, except in some particular cases where the structure of the tasks causes the GA to yield better solutions. To exploit the advantages of both algorithms, in [12] an algorithm that integrates them is presented.

The initial population for the GA is composed as follows: the result of executing the pragmatic algorithm is replicated ten times to obtain ten identical individuals; another 90 individuals are created from the previous individual, but this time some of the originally selected blocks are pseudo-randomly marked as non-selected. To complete the population, 100 more individuals are created according to the procedure termed as *i2*, including one special individual with all of the blocks selected.

4. EXPERIMENTAL RESULTS

To evaluate and compare the different algorithms, the real-time systems presented in [4] were used. The total number of systems is 28. There are 14 different task sets. The tasks are synthetic and may contain sequential code, loops, nested loops, if-then-else conditional structures and any combination of these.

From each task set two systems are generated; the difference between them lies in the task periods. In one of the systems, the periods have been adjusted so that the number of task preemptions is low and the system utilisation, when using a conventional cache memory, is about 40%. To distinguish those systems, the suffix L is used. In the other system, the periods have been adjusted so that the number of task preemptions is high and the system utilisation, when using a conventional cache memory, is larger than 90%. To denote those systems, the suffix H is used.

Using the same task set but different task periods allows assessing the behaviour of the algorithms in front of a wider range of cache size requirements: as the task interference is increased, a bigger cache is required to keep the system schedulable, as can be observed in the following tables.

Fetching an instruction from cache takes 1 cycle while fetching an instruction from main memory takes 10 cycles. The mapping function for the locking cache is only direct-mapping, because this one is the most restrictive for the locking cache.

In order to tune the algorithms parameters, several runs were performed. Regarding population size, in accordance with previous works, sizes between 50 and 300 individuals were tested. With respect to mutation probability, fifteen values have been tested, ranging from 0.1% to 50%.

Furthermore, the effects of the pseudo-random number generation routine as well as the value used for its seed have been evaluated by running 25 times every experiment. Although the solutions, considered as the set of selected main memory blocks to be locked, were different, the number of cache lines required to keep the system schedulable is the same, regardless of the seed value.

The genetic algorithm parameters are:
- Population size: 200 individuals.
- Termination condition: 5000 generations.
- Crossover probability: 100%.
- Mutation probability: 8%.

The major goal of the experiments is to decide which algorithm is able to obtain a schedulable real-time system while using the smallest cache size. Another goal is to determine whether the proposed algorithms can improve the results obtained when repeatedly using the original algorithm, as presented in [4]. These issues can be ascertained by means of the following tables and graphics.

Since the results for *GAi1s1m1* and *GAi1s2m1* were identical, in both Tables 1 and 2, their results were merged in the *GAi1sXm1* column.

Table 1 shows the minimum number of cache lines obtained for each algorithm. The first column shows the real-time system used. Regarding the pragmatic algorithm, just one version is included since in all of the cases both algorithms yield the same result. The last column (*Original*) corresponds to the results obtained when using the original algorithm presented in [4].

In those cases where improved results were obtained with respect to the original algorithm, the best result is highlighted using a boldface type. From these highlighted results it is possible to observe that: algorithms with initialization type *1*, that is *GAi1sXmX*, perform worse than the original algorithm; there is no algorithm, from the proposed ones, capable of always beating the original algorithm; algorithms *GAi2s1m2*, *GAi2s2m2* and *Prag-GA* perform better than the others.

In order to help determining which algorithm offers the best results, the rows in Table 2 show the number of cases in which the new algorithm obtains a better result than the original algorithm (Better than Original, BtO), the number of cases in which the new algorithm is in a tie with the original algorithm (Equal to Original, EtO), and the number of cases in which the new algorithm is beaten by the original algorithm (Worse than Original, WtO).

The last row shows, in percentage, the average reduction on the number of cache lines obtained when using the different algorithms when compared to using the original algorithm. A negative reduction indicates that in order to make the system schedulable, the corresponding algorithm needs more lines than the original one.

From Table 2 it can be seen that *GAi2s1m2*, *GAi2s2m2* and *Prag-GA* offer the best results, being able to defeat the original algorithm in more than 75% of the cases with a reduction in cache size that lies between 7% and 10%.

Algorithms *GAi2s1m1* and *GAi2s2m1* also perform better than the original one, but in a lower scale.

The remaining algorithms –GAs with initialisation type *1* and the pragmatic algorithm– exhibit results that are not any better than those from the original one. Furthermore, they produce worse results in the majority of the cases.

To ascertain whether the differences between the different selection and mutation policies, as shown in Table 2, are statistically significant, a multi-sample comparison has been made considering the number of lines obtained by each algorithm minus the number of lines obtained by the original algorithm. This allows isolating the effects of the existing differences in magnitude in cache size from experiment to experiment –as a way of example, experiment $6H$ needs around 3000 cache lines while experiment $8L$ needs no more than 711–. This comparison also includes the *Prag-GA* algorithm.

The result of this comparison is shown in Figure 1, which illustrates the LSD (Least Significant Differences) means and intervals [8] with a confidence level of 95%. In this figure it can be observed that there are three algorithms that stand out (have a more negative mean) and with statistically significant differences (show no overlapping with others): those which employ a type *2* mutation policy (*m2*) and the hybrid *Prag-GA* algorithm, as one could expect since it also uses policy *m2*. Hence, the selection policy has no effect in the results while initialisation type *i2* when combined with mutation type *m2* exhibit the best results.

Last but not least, to statistically verify that *GAi2sXm2*-type algorithms are really able to defeat the original algorithm, three null hypothesis tests have been made: t-test, sign-test and signed rank-test. In all three cases the answer was the same, indicating that there is a significant difference

Table 1: Minimum number of cache lines obtained when using the different algorithms with 28 different real-time systems

	Prag	*GAi1sXm1*	*GAi1s1m2*	*GAi1s2m2*	*GAi2s1m1*	*GAi2s1m2*	*GAi2s2m1*	*GAi2s2m2*	*Prag-GA*	*Original*
Ex1H	2374	2505	2461	2455	2504	**2370**	2430	**2370**	**2370**	2374
Ex1L	265	463	516	519	265	265	265	265	265	265
Ex2H	2931	3336	3235	3219	3341	2990	3274	2930	2927	2931
Ex2L	277	530	702	711	**277**	**277**	**277**	**277**	**277**	302
Ex3H	1587	1598	1584	1583	1600	1572	1583	1567	1567	1482
Ex3L	509	785	776	774	584	526	573	524	**509**	524
Ex4H	1563	1539	1293	1341	775	**725**	764	824	1087	821
Ex4L	1350	782	408	416	214	200	209	202	**196**	515
Ex5H	1891	2070	1944	1946	2081	**1597**	2018	1650	1709	1891
Ex5L	293	649	743	735	365	297	332	**289**	291	293
Ex6H	3016	3161	3038	3041	3177	**2936**	3144	**2936**	**2936**	3016
Ex6L	143	205	328	327	143	**142**	143	**142**	143	161
Ex7H	3485	3599	3538	3521	3608	3455	3610	**3454**	3455	3485
Ex7L	394	1584	1639	1649	**394**	**394**	**394**	**394**	**394**	396
Ex8H	1831	1855	1828	1829	1824	1736	1850	1734	**1731**	1831
Ex8L	469	686	711	711	465	**448**	470	452	469	467
Ex9H	2972	2967	2734	2735	2973	**2308**	2850	2319	2436	2972
Ex9L	187	674	925	786	221	**158**	187	**158**	**158**	162
Ex10H	2347	3424	2346	2348	2357	**2340**	2355	**2340**	2341	2346
Ex10L	399	586	605	621	**398**	**398**	**398**	**398**	**398**	402
Ex11H	1731	1812	1556	1555	1842	1178	1458	**1121**	1441	1718
Ex11L	250	107	129	132	**56**	**56**	**56**	**56**	**56**	60
Ex12H	2249	2334	2290	2289	2340	**2204**	2330	**2204**	2208	2249
Ex12L	353	722	754	765	364	353	370	**352**	**352**	353
Ex13H	757	767	750	752	772	**746**	762	**746**	752	757
Ex13L	20	20	22	22	20	20	20	20	20	20
Ex14H	1808	1998	1929	1918	1747	**1466**	1660	1506	1614	1809
Ex14L	1	1	1	1	1	1	1	1	1	4

Table 2: Table 2. Number of cases in which the new algorithms beat, tie, lose in front of the original algorithm and average percentage of reduction (if any)

	Prag	*GAi1sXm1*	*GAi1s1m2*	*GAi1s2m2*	*GAi2s1m1*	*GAi2s1m2*	*GAi2s2m1*	*GAi2s2m2*	*Prag-GA*
BtO	7	2	6	6	11	**21**	11	**23**	**23**
EtO	13	1	1	0	2	3	2	3	2
WtO	8	25	21	22	15	4	15	2	3
Red (%)	-17,7	-50,6	-58,4	-56,1	1,3	9,8	4,0	9,4	7,1

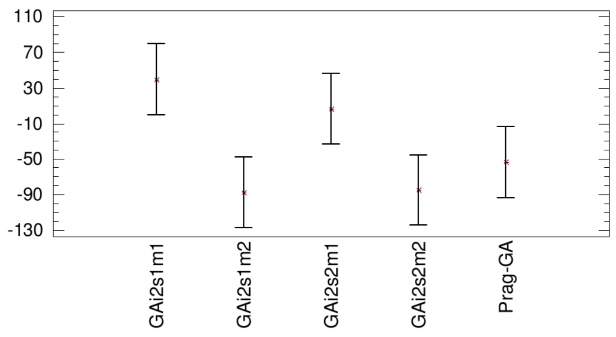

Figure 1: LSD Intervals for *GAi2sXmX* minus *Original* (confidence level 95%).

between the results obtained when using *GAi2sXm2*-type algorithms in front of the original algorithm.

Another parameter that has to be evaluated is the execution time of the algorithms. This time does not affect in any way either the performance of the final system or the cache size, but the system development time.

Since both the population size and the number of iterations for all of the algorithms is the same, their execution times are constant. For the *Prag-GA* however, which previously executes the pragmatic algorithm, it is necessary to account for one extra minute, which in the end, does not add much. Likewise, the differences in applying one mutation type or the other or a given selection policy are also non significant.

Therefore, five algorithms, (*Prag-GA* and the *GAi2sXmX* family) need on average eight minutes to finish, with a maximum of 12 and a minimum of five.

These variations in the execution time result from the disparity present in the amount of cache lines a given real-time system task needs: estimating the CRTA in an iterative way takes more or less time according to the degree of interference between the tasks. Also, the number of tasks in the system affects the time required to estimate the CRTA.

The original algorithm has an execution time quite similar to the new algorithms presented in this paper. There is however a major drawback: it is necessary to execute it in a repeated manner, testing with different cache sizes until the minimum is found. Therefore, when using the original algorithm in order to solve the problem, several dozens of executions may be required; in contrast, with the algorithms presented in this paper, one execution suffices.

To finish the analysis, the generation number in which every algorithm has found the best individual has also been recorded. Once more, this result has been strongly influenced by the tasks in the real-time system.

The result of a multi-sample comparison is shown in Figure 2 where the LSD intervals are presented. From this figure it can be seen that there is no difference in the convergence of the different algorithms, except for *GAi2s1m1*, which clearly needs much more iterations to obtain a poor result with respect to the other algorithms.

This is due to using the *s1* selection policy combined with the *m1* mutation policy, which establishes that the best individual in the current generation is copied in the next generation but may suffer mutation. This way, it is possible that

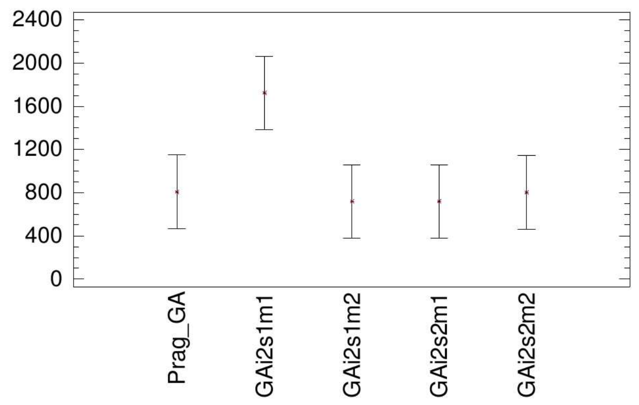

Figure 2: LSD Intervals for convergence iteration (confidence level 95%).

it gets more lines (mutation *m1*), causing the algorithm to go back; that is, having more lines as iterations advance.

For the latter algorithms, the mean value for the generation in which the best solution is found is around 800; in some of the experiments however, the best solution is reached around 3000 generations. Therefore, choosing 5000 generations as the stop condition seems to be adequate.

5. CONCLUSIONS

This paper presents a series of algorithms for selecting contents to be loaded into a locking cache in order to minimise the cache size so that a real-time system is schedulable. The locking cache memory is locked in a static manner, which means that once the system starts operating, there are no cache replacements. The systems analysed are pre-emptive, fixed-priority, multitasking real-time systems.

The starting point was a genetic algorithm developed to optimise the performance of real-time systems using a static approach to lock the cache memory that was then adapted to solve the cache size minimisation problem. This original algorithm works with pre-determined cache sizes; hence, to find the minimum cache size, it is necessary to make a high number of executions with different cache sizes. Besides, the number of executions was not delimited, since it depends on the characteristics of the real-time system such as number of tasks or relationship between task periods.

The objective of this work was developing new algorithms that satisfy two goals: First, to improve the solutions obtained previously. This means guaranteeing that the real-time system is still schedulable with smaller cache sizes. The second goal is to make just a single execution to get a minimum cache size within a bounded and small execution time.

In this work, ten algorithms have been developed. All of them output a set of instructions to be loaded and then locked into cache memory while guaranteeing that the real-time system is schedulable.

From these ten algorithms, five of them are not able to decrease the number of selected blocks compared against the original algorithm; in fact, they need bigger cache sizes.

These five algorithms are the pragmatic algorithm and the GAs that use initialisation type *1*. The problem with the former is that it chooses the blocks by using a static estimation of the worst-case execution path. That is, the estimation is made before the block selection is done and the worst-case execution path remains the same during the entire analysis.

Previous works however show that the worst-case execution path may vary as the blocks are loaded into cache memory. The pragmatic algorithm however can not adapt to these variations.

Regarding quality of the results obtained, the deciding factor is the initial population policy. The greater the variability of the individuals in the initial population, the better the results are.

In particular, for GAs with initialisation type *1*, since all the individuals are identical at the beginning, the contribution made by selection and crossover operators is negligible; thus the mutation is responsible for the evolution of the population. Unfortunately, the effect of the mutation operator is not strong enough to get good solutions.

The four algorithms with initialisation type *2* are able to improve the results obtained by the original algorithm, which indicates the importance of the policy used to create the initial population. Two of these algorithms, those using mutation type *2* –which always eliminates blocks–, consistently obtain the best results independently of the selection policy used, whose effect is negligible with respect to the quality of the solution.

On the other hand, the hybrid *Pragmatic-GA* algorithm, which uses mutation type *2* and initialisation type *2* to create a part of the initial population, exhibits identical behaviour to algorithms in the *GAi2sXm2* family. Therefore, having a good starting population does not necessarily guarantee obtaining better solutions.

Taking into account the results regarding both the quality of the solution and the execution times, the authors consider that the best algorithms are *GAi2s1m2* and *GAi2s2m2*. These algorithms provide a solution to the problem of selecting locking cache contents to get an schedulable hard real-time system, while keeping the cache memory in its minimal size. Also, their results are in almost all of the cases better than those provided by previous version of genetic and greedy algorithms.

6. ACKNOWLEDGMENTS

This work has been partly supported by the Universitat Politècnica de València project AID.

7. REFERENCES

[1] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 204 –212, jun 1996.

[2] A. M. Campoy, A. P. Jimenez, A. P. Ivars, and J. V. B. Mataix. Using genetic algorithms in content selection for locking-caches. In *IASTED International Symposia Applied Informatics*, pages 271 –276, feb 2001.

[3] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 49–56, Washington, DC, USA, 2005. IEEE Computer Society.

[4] A. M. Campoy, F. Rodriguez-Ballester, R. Ors, and J. Serrano. Saving cache memory using a locking cache in real-time systems. In *Proceedings of the 2009 International Conference on Computer Design*, pages 184–189, jul 2009.

[5] M. Campoy, A. P. Ivars, and J. V. B. Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium*, dec 2001.

[6] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 143–148, New York, NY, USA, 2007. ACM.

[7] Y. Liang and T. Mitra. Instruction cache locking using temporal reuse profile. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 344–349, New York, NY, USA, 2010. ACM.

[8] G. McPherson. *Applying and Interpreting Statistics. A Comprehensive Guide*. Springer Texts in Statistics. Springer, 2nd edition, 2001.

[9] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[10] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 1994.

[11] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proceedings of the 2nd International Workshop on worst-case execution time analysis, in conjunction with the 14th Euromicro Conference on Real-Time Systems*, jun 2002.

[12] I. Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[13] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS '02, pages 114–, Washington, DC, USA, 2002. IEEE Computer Society.

[14] A. Shaw. Reasoning about time in higher-level language software. *Software Engineering, IEEE Transactions on*, 15(7):875 –889, July 1989.

[15] E. Tamura, J. Busquets-Mataix, and A. Marti Campoy. A parallel genetic algorithm for locking-cache contents selection in pre-emptive real-time systems. *Epiciclos Scientific Journal*, 4(1):9 –26, 2005.

[16] E. Tamura, J. Busquets-Mataix, J. Serrano, and A. Marti Campoy. A comparison of three genetic algorithms for locking-cache contents selection in real-time systems. In B. Ribeiro, R. F. Albrecht, A. Dobnikar, D. W. Pearson, and N. C. Steele, editors, *Adaptive and Natural Computing Algorithms*, pages 462–465. Springer Vienna, 2005.

[17] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. *SIGMETRICS Perform. Eval. Rev.*, 31:272–282, June 2003.