A PARALLEL GENETIC ALGORITHM FOR LOCKING-CACHE CONTENTS SELECTION IN PRE-EMPTIVE REAL-TIME SYSTEMS

E. Tamura

Grupo de Automática y Robótica
Pontificia Universidad Javeriana - Cali, Colombia
Calle 18 118-250, Cali, Colombia

E-mail: tek@puj.edu.co

J. V. Busquets-Mataix, A. Martí Campoy

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

E-mail: {vbusque, amarti}@disca.upv.es

En los sistemas de tiempo real el que los resultados sean correctos depende del momento en que éstos se produzcan. Para explotar la velocidad de procesamiento de los procesadores contemporáneos, es necesario emplear memorias cache; sin embargo, esto conlleva a tiempos de ejecución no predecibles. Si bien es cierto que es posible obtener predecibilidad cuando los contenidos de la memoria cache no se modifican, la obtención de buenas prestaciones, demanda unas selección cuidadosa de las instrucciones que se cargarán en la memoria cache para no ser modificadas. Es en este aspecto, el de la resolución de problemas de optimización, donde los algoritmos genéticos han demostrado su valía. Adicionalmente, son fáciles de paralelizar. Este artículo discute el método, los criterios y los resultados obtenidos al paralelizar un algoritmo genético que permite seleccionar los contenidos de la memoria cache para su uso en sistemas de tiempo real.

In real-time systems, the correctness of its results is time dependent. To cope with the operational speed of contemporary processors, they may use cache memories, thus leading to unpredictable execution times. Predictability, however, can be obtained by locking the cache contents. Nevertheless, to yield good performance, it is necessary to select the instructions that will be locked into the cache. Genetic algorithms are very appropriate for solving this kind of optimisation problems. Furthermore, they are suitable candidates for parallelisation. This paper discusses the method, criteria and, the results from parallelising

a genetic algorithm for locking-cache contents selection in real-time systems.

Key Words: Genetic Algorithms, Real-Time Systems, Locking-Cache, Parallelisation, MPI.

Recibido: 7-septiembre-2005 Revisado: 1-diciembre-2005 Aceptado: 6-diciembre-2005

0. INTRODUCTION

Processor-based products are increasingly common everyday; from the manufacturers viewpoint they provide good price/versatility ratios and on the other hand, from the user viewpoint they provide better functionality. This situation obeys to the lower cost of mass-produced processing elements and follows a never-ending push-pull pattern. As users demand for more computing power elsewhere, the use of processing elements ranging from systems with simple 4-bit microcontrollers to 32- or even 64-bit processors pervades our life giving rise to ubiquitous computing. Some of these systems may demand very stringent timings to perform well; systems like these are called hard real-time systems. Many engineering systems incorporate a real-time system to control its operation in part or fully, giving rise to embedded real-time systems. In this kind of systems, time is absolutely critical, to the point that a system failure may cause severe consequences. Hence, designers must be completely sure that the system will perform as expected (as per the requirements).

To cope with the processing power required 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 be on time even under adverse circumstances. To do so, it is necessary to estimate the worst-case execution time of every task in the system.

To minimise the speed disparity between the memory system and contemporary processors, computer architects introduce cache memories in the memory subsystem, so that a memory hierarchy is created. A cache memory is a small, but fast memory that holds a fraction of the contents of the main memory, which is bigger but much more slower. The inclusion of a cache memory provides a better average performance in virtue of two principles: spatial and temporal locality of reference. The former takes into account the fact that for any task, the instructions in the vicinity of the currently executing instruction have a higher probability of execution than those which are farther apart. The latter implies that an instruction that is referenced at one point in time will have a higher probability of being referenced again sometime in the near future than other instructions that were referenced

earlier; as an example, instructions inside loops may get executed more times than others.

Cache memories are thus extensively used to reduce the increasing speed gap between processor and main memory because they provide lower average execution times by minimising the number of accesses to main memory. The cache controller tries to keep in the cache memory the most recently used instructions/data trying to exploit its locality of reference. General-purpose systems benefit directly in a transparent manner from this architectural improvement. In general, real-time systems do not; it is mandatory to account for the cache presence.

The remainder of the paper is organised as follows. Section 1. introduces the real-time aspects required to understand the rest of the paper. Section 2. explains the effects of including cache memories in a real-time system. Section 3. explains how locking caches basics. Section 4. introduces the problem and the algorithm to solve it. Section 5. illustrates the design criteria and some implementation details. Section 6. outlines the experimental design. Section 7. reports results and analysis. Section 8. presents some concluding remarks.

1. REAL-TIME BASICS

In a real-time system there are usually two kind of tasks: tasks that are in charge of controlling the environment in which they are embedded and tasks that usually do housekeeping chores. The former are termed real-time tasks, because they must satisfy some timing constraints dictated by the environment (the correctness of its results is time dependent); the latter, usually, are not necessary for correct operation, but do provide administrative functions, so it is desirable that from time to time there is computing power available to them.

A real-time system reacts to external events by following a three separate, but related phases: acquire the status of the external environment; determine the appropriate action to correct any deviations from desired operation; emit response to the environment. This cycle is repeated as long as the system is operating. As the system complexity grows, there are a lot of cycles akin to the one described previously. The system response must be bounded in time.

In order to determine whether the system will satisfy its timing constraints, the real-time systems designer must determine if the system is schedulable; that is, given the period, T, which is the time elapsed between two successive activations of a real-time task, and its deadline, D, which is the time limit given to a real-time task to finish its execution since its previous activation, it is indispensable to guarantee that every real-time task will finish its execution on time. To do so, it is necessary to estimate the worst-case response time of every task in the system. In a real-time system, most of the real-time tasks are periodic; aperiodic tasks are modelled as periodic tasks where the period is given by the minimum inter-arrival

period. Sporadic tasks run whenever there is time available. The purpose of the analyses is then to guarantee that every real-time task will finish its current cycle before some predetermined deadline. Non real-time tasks utilise the processing time that the real-time tasks do not use within its corresponding period.
Within the real-time tasks, not all of the tasks have the same urgency to execute. Hence, they are assigned priorities, which reflect how urgently the task needs the processor in order to satisfy the timing requirements imposed by the external environment. In general terms, the scheduling policy, which is responsible of determining which task to run when the processor is available, is influenced by task priorities. In particular, in preemptive real-time systems, the task with the highest priority will have the processor for as long as it wants. Furthermore, processor time is not allotted to any task with a lower priority as long as there is any task with higher priority that is ready to execute. On the other hand, whenever a high priority task demands the processor and it is being used by a lower priority task, the execution of the lower priority task is deferred and then the processor is allocated to the high priority task. This situation is termed *preemption.*
Hence, estimating the worst-case response time of the lowest priority task involves knowing the *Worst-Case Execution Time* (*WCET*) of the rest of the tasks in the system. In general, the *Worst-Case Response Time* (*WCRT*) for fixed-priority scheduling is estimated by means of *Response Time Analysis, RTA*, which is calculated by means of a recurrence relationship given by Joseph and Pandya [7]:

$$w_i^0 = C_i \quad (1)$$

$$w_i^{n+1} = C_i + \sum_{\forall \tau_j \in hp(\tau_i)} C_j \left\lceil \frac{w_i^n}{T_j} \right\rceil \quad (2)$$

Given a task τ_i, RTA works by considering the interferences produced by the execution of the higher priority tasks on τ_i within an increasing time window w_i^n where n is the recurrence index. The response time, R_i, of task τ_i is the fixed point of the sequence given in equation 2 where $hp(\tau_i)$ denotes the set of tasks that have a higher priority than τ_i. Notice that the equations assume tasks with distinct priorities.
In simple words, the worst-case response time of any task τ_i depends, not just on its own worst-case execution time, C_i, but also on the maximum number of preemptions that τ_i may suffer, which is a function of the worst-case execution time of every task τ_j whose priority is higher than that of τ_i.
For any task τ_i, this series converges to τ_i's response time, R_i, when $w_i^{n+1} = w_i^n$. R_i can then be compared against τ_i's deadline, D_i, to determine τ_i's schedulability. Notice that in this work, $D_i = T_i$.

2. REAL-TIME SYSTEMS AND CACHE MEMORIES

However, in the previous equation, cache effects are not involved. Modelling cache behaviour is very complex, as it has been described in several papers during the last decade: cache memories present a dynamic, adaptive and non-predictable behaviour thus making hard to predict the execution times since it is then possible that instruction cache contents are overwritten by other tasks instructions or even for some other instructions of the same task and thus the required schedulability analysis turns more complex when dealing with cache memories.

This lack of determinism precludes the use of well-known schedulability analysis techniques in order to validate the temporal correctness of the system. The use of cache memories in real-time systems arises two hard problems. The first problem is to calculate the WCET, due to *Intra-task* or *Intrinsic Interference* [1]. Intrinsic Interference occurs when a task removes its own instructions from the instruction cache (*I-cache*) due to conflict and capacity misses. When the removed instructions are referenced again, cache misses increase the execution time of the task. This way, the delay caused by the I-cache interference must be included in the WCET calculation. The second problem is to calculate the task response time due to *Inter-task* or *Extrinsic Interference* [3]. Extrinsic Interference occurs in preemptive multitasking systems when a task displaces instructions of any lower priority task from the I-cache. When the preempted task resumes execution, a burst of cache misses increases its execution time. This effect, called *Cache Refill Penalty* [4] or *Cache-Related Preemption Delay* [6], causes an extra delay, γ_j^i, that must be considered in the schedulability analysis, since it situates task execution time over the pre-calculated WCET.

To do so, it is necessary to introduce an extra term in Eq. (2), which takes into account the penalties suffered by the tasks when executing concurrently. The estimation is termed *Cached Response Time Analysis, CRTA* [5], and here, Eq. (2) turns out into

$$w_i^{n+1} = C_i + \sum_{\forall \tau_j \in hp(\tau_i)} (C_j + \gamma_j^i) \left\lceil \frac{w_i^n}{T_j} \right\rceil \quad (3)$$

The difficulty in analysing the worst-case response times resides in the fact that it is necessary to know which instructions are loaded in cache memory in every moment. Another possibility to simplify the analysis is to determine a priori the instructions to be loaded in the cache memory, and then lock its contents (i.e. disable cache replacement) so that the interference is reduced to a minimum. One such approach, proposed by Martí et al. [10], that takes advantage of the performance of cache memories while achieving predictability is to use a locking cache.

3. LOCKING-CACHE BASICS

A *locking cache* is a cache memory with the ability to lock its contents; that is, to preclude that they are replaced. It operates by following three steps:

1. Select main memory blocks to load;
2. Load them into the locking cache; and
3. Lock them.

Several processors include *locking instruction caches* (*locking I-caches*) to store critical-timing code, thus precluding its replacement as the processor fetches new instructions. The use of locking I-caches in real-time systems offers several advantages:

- Intrinsic interference is eliminated, and extrinsic interference is bounded and can be estimated in advance. This makes I-cache behaviour very predictable, thus allowing a simple analysis, even when other architectural features are used, since memory access delays are constant.
- In several cases, the use of locking I-caches effects the same or better performance than using a conventional I-cache.
- The use of a locking I-cache is transparent to end programmers, since he/she does need to neither include any special instructions nor use additional tools to write the applications.
- Last, but not least, locking caches are already incorporated in several embedded processors (e.g. ARM 940, ColdFire MCF5249, PowerPC 603e).

In a real-time system a locking cache can be used either in a static manner[11] or in a dynamic manner[13]. In both cases, the goal is to achieve determinism as well as good performance. The dynamic case however sacrifices determinism a little in order to obtain a higher performance in front of the static case.
In any case, no matter which way of using the locking cache is chosen, the performance of the real-time system absolutely rests on the instructions locked in the locking cache memory. Selecting at random the instructions to be locked provides determinism, but the resulting performance may not be comparable with that obtained when using a conventional cache memory. Therefore, a careful instruction selection is crucial to accomplish the aforementioned goal.

4. SELECTING LOCKING-CACHE CONTENTS

Choosing the cache contents in a way that maximises the probability of finding the instruction in cache is a combinatorial problem, where the number of combinations may be very huge and hence exhaustive search is infeasible. Petrank and Rawitz [16] have been proved that 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. In general, the techniques employed to solve combinatorial problems are characterised by looking for a solution from among many potential solutions. Furthermore, rather than trying to find only the best (optimal) solution, a good non-optimal (trade-off) solution is sought. Therefore, to solve the problem at hand, it is necessary to resort to some form of directed search. For this kind of problem, one of the most appealing techniques is using GAs since they are generally seen as optimization methods for non-linear functions.

GAs, proposed by Holland [9], have their theoretical roots in biology and attempt to mimic the process of natural selection, discovered by Charles Darwin. In general, a GA starts from an initial generation of individuals containing random values and then creates successive generations from the first by applying the standard rules of natural evolution: natural selection, mating (crossover), and mutation. Natural selection heuristically privileges the reproduction of the best individuals: those who best fit into the environment (survival of the fittest). Thus, the evolution process, evolving through the generations, tends to produce individuals with a better fitness, i.e. those that tend to maximize the fitness function.

In fact, the experimental results presented by Martí et al. [12] verify that the use of a genetic algorithm to solve the problem represents a good choice since it provides not just the set of instructions to be loaded but also because it offers good performance. Furthermore, in a more recent work, Martí et al. [14] compare the performance of two algorithms for static locking of I-caches: one is the aforementioned genetic algorithm; the other is a pragmatic, reference-based algorithm from Puaut and Decotigny [17], which uses the string of memory references issued by a task on its worst-case execution path as an input to the cache contents selection algorithm. A direct-mapped cache memory organisation with sizes ranging from 1 KB to 64 KB was used for the experiments.

Experimental results show that (i) both algorithms behave identically with respect to the worst-case system utilisation; (ii) the genetic algorithm behaves slightly better than the reference-based algorithm with respect to the average slack of tasks. In fact, the former can be tailored to improve on a particular metric just by adjusting the fitness function. On the other hand, the latter, which is optimised to improve the system utilisation, would need more rework; and (iv) the execution time of the cache contents selection procedure is much better when using the reference-based algorithm than with the genetic algorithm.

Therefore, since executing the genetic algorithm takes much longer than running the reference-based algorithm, it is necessary to find a way to optimise its execution time. In the former case, arriving at a high quality solution involves a very large number of evaluations and consequently is computationally demanding. Fortunately, the intrinsic parallel nature of GAs makes them suitable for a parallel

implementation. According to Hart et al. [8], by parallelising a GA, it is possible to:

- Reduce the time to locate a solution (faster algorithms),
- Reduce the number of function evaluations (cost of the search),
- Have larger populations thanks to the parallel platforms used for running the algorithms, and
- Improve the quality of the solutions worked out.

The ultimate goal is to achieve a faster, low-cost, *Parallel Genetic Algorithm (PGA)*, relative to the sequential GA by taking full advantage of the inherent parallelism in GAs while keeping the quality of the result. To accomplish this goal, on one hand, the population needs to contain a diverse set of candidate solutions to get a better coverage of the search space, and on the other hand, the cost of evaluating the population fitness should be lower. These two goals can be achieved by ensuring a balanced distribution of work among nodes; reducing the amount of inter-processor communication, which is expensive; and, keeping low the overhead of communication and synchronisation. This is translated into partitioning the population in several sub-populations (*demes*), which evolve simultaneously in an independent manner; from time to time some individuals exchange its genetic code through cloning and migration to inject new diversity into converging demes.
Albeit a GA is able to converge without any knowledge of the problem, it is necessary to set some parameters such as the crossover and mutation probabilities and, the population size; it is also necessary to set some problem-specific parameters such as number of cache lines. Besides these parameters, the PGA also requires to know the number and size of the demes; the exchange policies (how to choose emigrants and how to integrate immigrants); the exchange rate (number of emigrants); the exchange pattern (where to migrate); and the exchange frequency (how frequent migrations are).
The programming style used for the PGA is known as *Single-Program-Multiple-Data, SPMD*, where every process executes the same code image, but with a different subpopulation. Besides that, the parameters that control the evolution of the subpopulations in each deme are the same, so the evolution policy applies the same selective pressure to each subpopulation. Hence, the parallelisation method is known as a homogeneous island approach, a coarse grain model.
Suitable values for the majority of parameters were found by trial and error and are fixed: number of generations is 800; global population size is 240 individuals; probability of crossover is 0.64; probability of mutation is 0.01, exchange frequency is one each 10 generations (one epoch). The size of the subpopulations is dependent on the number of nodes: the global population is evenly divided between them. The parameters concerning the PGA are consistent with those generally accepted as reflected in Alba and Tomasini's review [2].

The algorithm has three main phases: the Initialisation Phase, the Evolution Phase, and the Tournament Phase. Each of these phases contains one or more tasks, as illustrated in the following algorithm:

```
program SelectCacheContents
  const MaxGen = 800;

  /* INITIALISATION PHASE */
  StartUpMPI ();
  CurGen := 0;
  FittestIndiv := none;
  InitialisePopulation ( CurGen );
  /* EVOLUTION PHASE */
  while ( CurGen < MaxGen ) do
    /* Evaluation and Ranking */
    EvaluateAndRankPopulation ( CurGen );
    /* Preserve Fittest Individual */
    winner := SelectBestIndividual ( CurGen );
    FittestIndiv := Select ( FittestIndiv, winner );
    /* Migration */
    emigrant := SelectAndCloneIndividual ( CurGen );
    AsynchSend ( emigrant );
    /* Crossover */
    SelectParents ( CurGen );
    Offspring := MateParents ( CurGen );
    /* Immigration */
    AsynchReceive ( immigrant );
    /* Integration */
    IntegrateImmigrant ( Offspring, immigrant );
    /* Mutation */
    CurGen := ApplyMutation ( Offspring );
    CurGen := CurGen + 1;
  end while;
  /* TOURNAMENT PHASE */
  FittestIndiv := BinaryTournamentSel ( FittestIndiv );
end program;
```

The PGA works as follows:

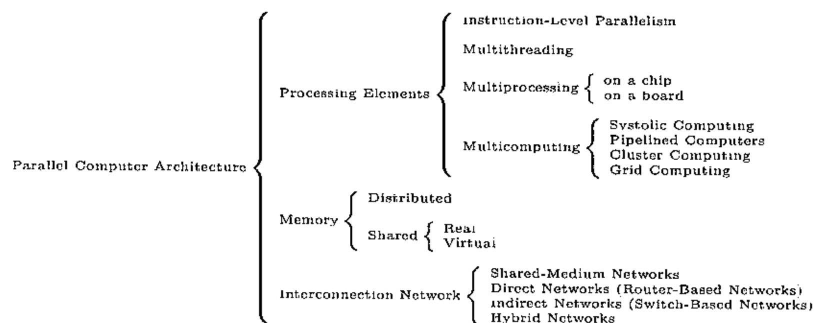
1. Each separate processing element independently generates its own initial semi-isolated random subpopulation (deme). This process of initial random creation takes place in parallel at each processing element. As soon as each separate processing element finishes this one-time task, it begins the Evolution Phase loop.

2. In the Evolution Phase, the task of measuring the fitness of each individual is first performed locally at each processing element. Then, the individuals are ranked according to a given fitness function; this selection step is performed locally at each processing element.
3. One individual in each subpopulation is selected at random for emigration from each processing element to other processing elements, giving rise to *speciation*, to introduce new breeds into each subpopulation.
4. Crossover is performed locally at each processing element. The processing elements operate asynchronously in the sense that each generation starts and ends independently at each processing element. Because each of these tasks is performed independently at each processing element and because the processing elements are not synchronised, this asynchronous island approach to parallelisation efficiently uses all the processing power of each processing element.
5. Previous to the stage of mutation (or better yet, as a part of it) immigration takes place. It is done asynchronously, so that the algorithm does not stop to wait for any slow processing element.
6. Once the termination conditions in every processing element have been satisfied, in order to choose the best individual, a tree-like tournament selection will be used, requiring at most $\log_2 P$ steps, where P is the number of processing elements.

The PGA provides, for every task in the task set, the subset of blocks to be locked. It also brings an estimation of the WCET of each task with the chosen set of blocks already loaded, and the WCRT of all tasks considering the estimated WCET.

5. DESIGN CRITERIA AND SOME IMPLEMENTATION DETAILS

On the hardware side, there are three essential components in a parallel architecture. Each component in turn has several categories:



By analysing the problem characteristics, the multicomputing environment, given its greater computing power, represents the best alternative. In fact since a cluster of PCs uses off-the-shelf PCs, it seems to be the most appropriate choice given its availability, nice scalability on computational resources (processing element, memory), and last but not least, affordability.

Once this selection has been made, the most straightforward choice for the memory system is using a distributed memory with a *NORMA* (*NO Remote access Memory Architecture*) architecture.

Regarding the Interconnection Network, a switched-based topology represents a good choice. A regular topology with a Multistage Interconnection using 10Base-X-based, off-the-shelf components, may be more than sufficient, since it provides an affordable and scalable interconnection network.

On the software side, the decision is quite easy, given the hardware platform chosen: use a message passing approach.

Both the Linux operating system and MPI are de facto standards, are been actively developed (which means, extensively debugged), are widely available and portable (with several distributions for many platforms freely available), and the learning curve is not too steep.

Summarising, on the hardware side, workstation clusters are well adapted for the implementation of this model. Thus, participating nodes in the search for good solutions work in a cooperative and decentralised manner. On the software side, MPI, a very portable, widely used and freely available message-passing model, perfectly suits the model.

The PGA is implemented in C and it was compiled using the Portland Group, Inc. C compiler provided with the CDK 4.1.2 and then linked against the Myricom implementation of MPI (based on MPICH) libraries. The application executes on a Xeon™-powered IBM e-server®Cluster 1350 running Red Hat Linux 7.3. The nodes are interconnected via Myrinet™-2000 from Myricom.

A simple static assignment reduces to zero the extra work required to determine and manage a good assignment: each MPI process is mapped to a different processing node. Therefore, at run time, there is no management overhead involved.

In a message-passing model, like MPI, the program manages data movement between a remote node and the local node explicitly. Since local memory accesses provide higher bandwidth and lower latency, data access performance can be improved by improving the data locality in the program. In the PGA, most of the data each process needs is stored in its own local memory, so during the evolution phase, the inter-processor communication volume is merely due to migrations. However, the data structure of an individual is quite complex (about 320 Kbytes), so migrating the individual as it is poses a costly communication overhead. Hence,

during the evolution phase, only the essential attributes of the individual (which demand just 4112 bytes) are migrated and then the rest of the information is reconstructed upon arrival at the destination. During the tournament phase, the data transmitted per sent message increases to 4356 bytes. In both phases, using this approach takes full advantage of the Myrinet MTU (9000 bytes) thus contributing to keep the communication-to-computation ratio very low.

To balance the communication bandwidth and also the computational load between the participating nodes the exchange pattern follows a bidirectional ring topology, which means that emigrants are only able to reach demes in its nearest neighbourhood (those located immediately to the left and right of the deme from which the emigrant departed). Since the code has no critical sections, there is no need for mutual exclusion operations. Then, to minimise synchronisation overheads, the communication between nodes is asynchronous.

To keep diversity and avoid conquering the other demes by applying too much selective pressure, the maximum exchange rate is one individual and the exchange policies establish that the emigrant is randomly chosen and that the new immigrant replaces a randomly chosen individual. The only synchronisation overhead takes place during the tournament phase, when the fittest individual amongst all the subpopulations is chosen through a binary tournament selection similar to the one presented by Pacheco [15]. However, since the selection requires a logarithmic number of steps (on the number of processing nodes) to arrive to the final solution, its overhead is not too high.

Both pack/unpack routines and datatype constructors [15, 18] were used in the MPI implementation. In general, there were no significant differences in execution time during the evolution phase. During the tournament phase, however using datatype constructors exhibited better timings than pack/unpack routines due to the scattered nature of the data required to send.

6. EXPERIMENTAL DESIGN

The goals here are: (i) to verify that the PGA is able to select the subset of memory blocks to be locked in less time than the sequential GA, and (ii) to determine whether the blocks selected by the PGA provide to the real-time system a better performance than the blocks selected by the sequential GA.

Therefore, experiments were conducted to determine the reduction in execution time and the quality of the solution. The (P)GA job is to select cache contents for different cache sizes ranging from 64 to 4096 cache lines and task sets with between 3 to 8 tasks each. Tasks have the usual statements found in any program. The whole set has 123 different experiments and it was executed on 1, 2, 4, 6, 8, and 10 nodes.

To measure the quality of the solution, Processor Utilisation, a commonly used metric to evaluate real-time systems performance was used. The lower the processor utilisation, the better, since this means that the task set demands less CPU time and thus other tasks might be included in the task set while the system remains schedulable (i.e., all tasks executing on time).
The execution on a single processor is strictly non-parallel; i.e. it does not have any MPI function call. Both the results of the execution and its execution times are averaged over the whole set to get statistically significant values. In spite of the microsecond accuracy of `MPI_Wtime`, execution time was measured with the `time` Unix command to time the overall run of the code. To take into consideration the inherent communications in the PGA, the datum corresponding to the `real` output, which measures the elapsed time between invocation and termination, was used.

7. RESULTS AND ANALYSIS

Since Processor Utilisation depends upon the task set and the number of cache lines being evaluated, by normalising Processor Utilisation against the one achieved by executing the GA in one processor it is possible to get a fairer metric. Table 1 shows the results of such normalisation.

Table 1
Comparison of Normalised Processor Utilisation

	2 nodes	4 nodes	6 nodes	8 nodes	10 nodes
Improve	84	107	104	104	101
	68,29%	86,99%	84,55%	84,55%	82,11%
Worsen	29	13	12	12	15
	23,58%	10,57%	9,76%	9,76%	12,20%

As it can be seen, it seems than using more than 2 nodes results in improvements in about 84.55% of the cases. By using Fisher's Least Significant Differences (LSD), it is possible to see (in Fig. 1) that, in fact, using 4, 6, 8 or 10 nodes gives statistically the same results with a 95% of confidence.

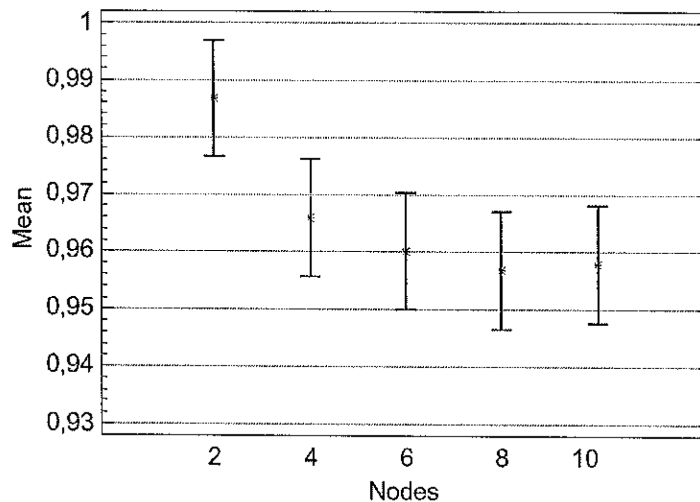


Figure 1. Means and 95,0% LSD Intervals

From Fig. 1 it is difficult to see however if using two nodes would give the same results. To solve the question, three null hypothesis tests concerning the body of the data for 2 and 4 nodes were used.

The first test is a t-test of the null hypothesis that the mean 2nodes-4nodes equals 0.0. The second test is a sign test of the null hypothesis that the median 2nodes-4nodes equals 0.0; it is based on counting the number of values above and below the hypothesized median. The third test is a signed rank test of the null hypothesis that the median 2nodes-4nodes equals 0.0; it is based on comparing the average ranks of values above and below the hypothesized median. All the three tests were rejected at the 95.0% confidence level.

To extract some statistics about execution time, profiling was done on several experiments using the PGI profiler. The results were then averaged and are as follows: Evaluation and Ranking occupy about 50% of the execution time, followed by Mutation (about 27%) and Crossover (about 13%). In total, they represent more than 90% of the execution time.

Most of the execution time is devoted to evaluation and ranking. Since the computational time is proportional to the number of cache lines and number of generations, substantial reductions in complexity were achieved by partitioning the global population and dealing with the resulting subpopulations in parallel, thanks to the computational power available.

The upper plot in Fig. 2 (left side) shows the projected execution time. It was drawn by taking the execution time in a single processor and then taking the quotient of this time against the number of processors. Therefore, it represents the idealised execution time for any number of processing nodes in the figure.

The line with triangles plots the observed mean execution time. As can be seen very clearly, it is below the idealised execution time. This can be explained as follows: On the software side, the plot with diamonds assumes a linear complexity in the code. However, the ranking task, which demands roughly 25% of the computing time, has a logarithmic complexity. On the hardware side, in addition to having more computational power, using more nodes also contributes with more cache memory, thus diminishing the traffic to local main memory when compared to a sequential execution in a single node. Thus, the combined effect of less main memory traffic and non-linear execution time per node leads to the exposed reduction in execution time. The two plots however are quite close. Hence, to observe the goodness of the proposed solution it is better to use another metric.

Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with P identical processors.

Measuring speedup of non-deterministic code is difficult; the definition used here is given by Alba and Tomasini [2] and it is known as *weak speedup with predefined effort*, in which the speedup ratio is computed by comparing the sequential and parallel version of the same algorithm using a predefined global number of iterations. In the general case, this metric is not considered as fair and meaningful as others because the algorithms are working out solutions of different fitness (quality). However, as shown before, results of the 123 experiments conducted show that the PGA solutions are of a slightly better statistical quality than that generated by the sequential version. Hence, in this particular case, we judge the metric as appropriate. Figure Fig. 2 (right side) shows the measured speedup.

Notice that the Expected Speedup has a 45° slope, which means that the execution time when using P processors, T_P, is given by $T_S \div P$. In other words, a slope of 45° means that as we double the number of processors, the time employed to solve the problem will be cut in half. Fig. 2 reveals that when the proposed PGA uses 4 or 8 processing elements it makes an excellent use of the computational resources. In fact, as can be seen, when using 4 processors, the PGA is about 5.55 times faster than the sequential GA; when using 8 processors, the PGA is about 13.84 times faster than the sequential GA. Notice however that even though using 10 processors provides a higher speedup than the expected one, there is a substantial reduction in speedup (the PGA is about 12.66 times faster than the sequential GA). Hence, using more than 8 processors with the PGA does not make an efficient use of the computational platform.

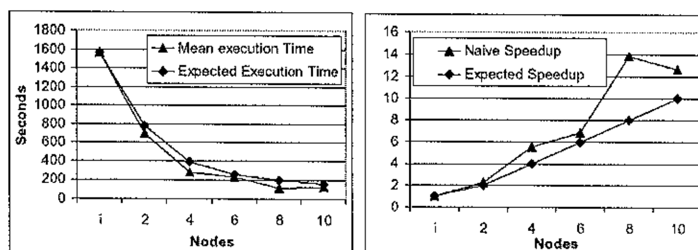


Figure 2. Observed mean execution time and Observed weak speedup with predefined effort

8. CONCLUDING REMARKS

Since tasks in the PGA are performed independently at each node and because the nodes are not synchronised, this homogeneous, asynchronous island approach to parallelisation, efficiently uses all the processing power of each node. Partitioning also favours a broader coverage of the search space and maybe a faster convergence due to simultaneous and cooperative search.
By virtue of this, the PGA neither needs to use a too large population nor too many processing nodes. Given that for 10 or less processing nodes, the calculation time represents a large percentage of the total execution time, the PGA not only performed much better from the point of view of computing times (w.r.t. the GA), which was expected; but it also converged to a solution of the same statistically quality with less computational effort. Furthermore, in this particular case, better cost-benefit ratios were achieved when using 4 to 8 processing nodes.
The island model with migration is an effective way to select the locking cache contents since it fully utilise the computing power of each node at all times: each processor concurrently begins its main generational loop as soon as it finishes its initial random creation of individuals at the beginning of the run; during the Evolution Phase, there is no need to synchronise the generations between nodes. Hence, each processor moves on to its next generation, regardless of the status of any other processor; the only phase during which an explicit synchronisation is required is the Tournament Phase.
In summary, the PGA is able to select blocks for the locking cache in a way that (i) together with the use of a locking cache provides predictable timing analyses; (ii) it provides similar processor utilisations than the sequential GA and the pragmatic algorithm given in ; and (iii) outperforms the execution time of the sequential GA.
In a future work, it will be very interesting to explore how different evolution policies in each deme impact the execution time and the quality of the solution.

REFERENCES

1. A. Agarwal, M. Horowitz, J. Hennessy, An Analytical Cache Model, ACM Theory of Computing Systems, vol. 7, no. 2, pp. 184–215, 1989.
2. E. Alba, M. Tomassini, Parallelism and Evolutionary Algorithms, IEEE Transactions on Evolutionary Computation, vol. 6, no. 5, pp. 443–462, 2002.
3. S. Basumallick, K. D. Nilsen, Cache Issues in Real-Time Systems, in: Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems, June 1994.
4. J. V. Busquets-Mataix, J. J. Serrano, The Impact of Extrinsic Cache Performance on Predictability of Real-Time Systems, in: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications, Tokyo, Japan, pp. 8–15, October 1995.
5. J. V. Busquets-Mataix, A. J. Wellings, J. J. Serrano, R. Ors, P. Gil, 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, L'Aquila, Italy, pp. 271–276, June 1996.
6. C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, C. S. Kim, Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling, in: Proceedings of the 17th IEEE Real-Time Systems Symposium, Washington D.C., USA, pp. 264–274, December 1996.
7. M. Joseph, P. K. Pandya, Finding Response Times in a Real-Time System, The Computer Journal, vol. 29, no. 5, pp. 390–395, 1986.
8. W. E. Hart, S. Baden, R. K. Belew, S. Kohn, Analysis of the numerical effects of parallelism on a parallel genetic algorithm, in: Proceedings of the Workshop on Solving Combinatorial Optimization Problems in Parallel. Cited in E. Alba, J. M. Troya, Improving flexibility and efficiency by adding parallelism to genetic algorithms, Statistics and Computing, vol. 12, pp. 91–114, 2002.
9. J. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, 1975.
10. A. Martí Campoy, A. Perles Ivars, J. V. Busquets Mataix, Using Locking Caches In Preemptive Real-Time Systems, in: Proceedings of the 12th IEEE Real-Time Congress on Nuclear and Plasma Sciences, IEEE Computer Society, Valencia, Spain, pp. 157–159, 2001.
11. A. Martí, A. Perles, J. V. Busquets Mataix, Static Use of Locking Caches in Multitask Preemptive Real-Time Systems, in: IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the 22nd IEEE Real-Time Systems Symposium), 2001.
12. A. Martí Campoy, A. Pérez Jiménez, A. Perles Ivars, J. V. Busquets Mataix, Using Genetic Algorithms in Content Selection for Locking-Caches, in: Proceedings of the IASTED International Symposia Applied Informatics, Acta Press, pp. 271–276, 2001.
13. A. Martí, A. Perles, J. V. Busquets Mataix, 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, 2002.
14. A. Martí Campoy, I. Puaut, A. Perles Ivars, J. V. Busquets Mataix, Cache Contents Selection for Statically-Locked Instruction Caches: an Algorithm Comparison, in: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05), Palma de Mallorca, Spain, pp. 49–56, July 2005.
15. P. Pacheco, Parallel Programming with MPI, Morgan Kaufmann Publishers, 1996.
16. E. Petrank, D. Rawitz, The harness of cache conscious data placement, in: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 101–102, 2002.
17. I. Puaut, 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 (RTSS02), Austin, USA, pp. 114–123, December 2002.

18. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI - The Complete Reference. Volume 1, The MPI Core, second edition, The MIT Press, 1998.