Proceedings of the First IEEE Conference on Evolutionary Computation - IEEE World Congress on Computational Intelligence, June 26-July 2, 1994, Orlando, Florida, pp. 829-833.

Dynamic Scheduling of Computer Tasks Using Genetic Algorithms

Carlos Alberto Gonzalez Pico
Roger L. Wainwright

Abstract— **In this paper we concentrate on non-preemptive hard real-time scheduling algorithms. We compare FIFO, EDLF, SRTF and genetic algorithms for solving this problem. The objective of the scheduling algorithm is to dynamically schedule as many tasks as possible such that each task meets its execution deadline, while minimizing the total delay time of all of the tasks. We present a MicroGA that uses a small population size of 10 chromosomes, running for 10 trials using a rather high mutation rate with a sliding window of 10 tasks. The steady-state GA was determined to be better than the generational GA for our MicroGA. We also present a parallel MicroGA model designed for parallel processors. The parallel MicroGA works bests when migration is used to move tasks from one processor to another to even out the load as much a possible. Test cases show that the sequential MicroGA model and the parallel MicroGA model produced superior task schedules compared to other algorithms tested.**

I. INTRODUCTION

The scheduling of industrial production processes is an economically important task. However, it is also a computationally difficult task. The scheduling problem is one of the classical computer science problems, and has been shown to be NP-complete [9]. This means there is no known polynomial time algorithm to optimally solve this problem. Instead researchers have relied on near optimal heuristic techniques for searching through the huge problem space. These techniques include several well known greedy algorithms such as First-In-First-Out (FIFO), Earliest-Deadline-First (EDLF), and Shortest-Runtime-First (SRTF). Other near optimal techniques include simulated annealing and genetic algorithms.

There are two types of real-time scheduling systems, soft real-time systems and hard real-time systems. In soft real-time systems, tasks are performed by the system as fast as possible. However, the tasks are not constrained to finish by a specific time [13]. In a hard real-time system, tasks have to be performed in a timely fashion where task deadlines are honored as much as possible. There are two types of hard real-time scheduling systems, static and dynamic. A static approach calculates schedules off-line and requires a complete *a priori* knowledge of the characteristics of all of the tasks [17]. A dynamic system determines a schedule for tasks "on the fly" as they arrive, and attempts to insure as best as possible that each task meets its deadline. Tasks can either be preemptive or non-preemptive. A preemptive task can be interrupted once it has started to execute. A non-preemptive task will be allowed to run until completion once it has been started. A non-preemptive scheduling is much more difficult to optimize than a preemptive scheduling.

This research has been partially supported by OCAST Grant AR2-004.
The authors are with the Department of Mathematical and Computer Sciences, The University of Tulsa, 600 South College Avenue, Tulsa, Oklahoma 74104-3189

In this paper we will concentrate on non-preemptive hard real-time scheduling algorithms. We compare FIFO, EDLF, SRTF and genetic algorithms for solving this problem. The objective of the scheduling algorithm is to dynamically schedule as many tasks as possible such that each task meets its execution deadline, while minimizing the total delay time of all of the tasks. In Section II a brief overview of genetic algorithms is presented including our genetic algorithm implementation for this problem. In Section III tests cases are described comparing all of the algorithms. Our MicroGA for a sequential scheduling model and our MicroGA parallel scheduling model are presented in Section IV.

II. GENETIC ALGORITHM IMPLEMENTATION

Genetic Algorithms (GA) are applicable to a wide variety of problems. In particular, genetic algorithms have been very successful in obtaining near-optimal solutions to many different combinatorial optimization problems [2,4,6,7,15,19]. Genetic Algorithms are based on the principles of natural genetics and survival of the fittest. Genetic algorithms search for solutions by emulating biological selection and reproduction. In a GA the parameters of the model to be optimized are encoded into a finite length string, usually a string of bits. Each parameter is represented by a portion of the string. The string is called a chromosome, and each bit is called a gene. Each string is given a measure of "fitness" by the fitness function, sometimes called the objective or evaluation function. The fitness of a chromosome determines its ability to survive and reproduce offspring. The "least fit" or weakest chromosomes of the population are displaced by more fit chromosomes.

The genetic algorithm is a robust search and optimization technique using probabilistic rules to evolve a population from one generation to the next. The transition rules going from one generation to the next are called genetic recombination operators. These include Reproduction (of the more "fit" chromosomes), Crossover, where portions of two chromosomes are exchanged in some manner, and Mutation. Crossover combines the "fittest" chromosomes and passes superior genes to the next generation, thus providing new points in the solution space. Mutation is performed infrequently. A new individual (point in the solution space) is created by altering one of the bits of an individual. Mutation ensures the entire state space will eventually be searched (given enough time), and can lead the population out of a local minima.

In a generational GA the offspring are saved in a separate pool until the pool size is reached. Then the children's pool replaces the parent's pool for the next generation. In a steady-state GA the offspring and parents occupy the same pool. Each time an offspring is generated it is placed into the pool, and the weakest chromosome is "dropped off" the pool. These two cases represent two extremes in pool management for genetic algorithms. It is assumed the reader is generally familiar with the fundamentals of genetic algorithms. Genetic algorithm packages for a single processor have been available for several years. The research reported here made use of LibGA [5], a GA package developed in house. Davis, Goldberg, and Rawling provide an excellent in depth study of genetic algorithms [6,10,16].

It is natural to represent a schedule of tasks in a chromosome as a permutation of the list of tasks known to the system at the time. Each gene represents a task, and the corresponding order of tasks determines the schedule. Each task has an associated run-time and deadline time. In our model we have developed what we call a *Sliding Window Approach* for processing the tasks. We define a fixed window size and only consider those tasks within the window.

Since the processing must be done in real-time we use a small population size and run the GA for a short amount of time. Our results have shown that a steady-state GA works much better than a generational GA for this type of environment. In our model, the GA works on a small population of schedules (say 10 chromosomes) for a small fixed number of trials, say 10. After 10 trials the first task (gene) in the best chromosome in the population is scheduled for execution. Then one additional task is obtained from the external task queue to restore the number of tasks under consideration back up to 10. In each chromosome in the population the scheduled task is replaced by the new task. In this way information gained in the previous GA run can be carried over to the next GA execution, rather than generating an initial random population. The GA works on the new list of tasks to determine the next task to schedule. Hence, we define a sliding window with a width of 10 tasks to slide through the entire list of tasks one at a time, simulating a real-time scheduling environment. We arbitrarily picked the window size of 10 and the number of trials at 10 to simulate the small amount of time that can be devoted to the GA before the next task will need to be scheduled. We call this GA a MicroGA. We define the delay of a task as the deadline - completion time. Notice Delay can be either positive or negative. Of course, a positive delay is desired. To penalize a schedule (chromosome) for negative delays and a reward a schedule for the number of completed tasks, we defined our evaluation function as the sum of all of the negative delays of the tasks executed + (Number of Completed Tasks * 100). The factor of 100 is arbitrary and can be altered if a researcher wants to weight task delays versus number of completed tasks differently. Notice, the evaluation function may be negative of positive. There is no significance to a positive fitness value over a negative fitness value, except that the larger the fitness value the better the schedule.

The use of Genetic algorithms in computer task scheduling has not been studied extensively. Several researchers, however, have used GAs in scheduling computer tasks. Bruns [3] introduced the idea of a knowledge augmented genetic algorithm for the solution of a real world production scheduling problem. Kanet and Sridharah [11] developed a genetic algorithm called PROGENITOR for production scheduling. However, PROGENITOR does not work well for real-time systems. Falkenauer and Bouffouix [8] developed the idea of using a special crossover operator called Linear Order Crossover (LOX) for their implementation of the Job Shop Scheduling (JSS) problem. Biegel and Davern [1] also developed several techniques for the JSS problem. Other research in computer task scheduling include [12,14,18].

III. TEST CASES

We have defined three datasets to test our algorithms. Each dataset represents a sequence of ordered pairs of tasks containing deadline and run-time values. We designed a set of 80 tasks that can run with no delay if the EDLF algorithm were executed. Hence an optimal schedule of 80 completed tasks without delay is possible for this dataset. We call this dataset the *Contrived dataset.*

The tasks in the Contrived dataset were randomized before being applied to our model. The second dataset is called *Random Dataset I* where 80 tasks were generated with deadlines uniformly random from zero to 5000, and the range of run times were generated uniformly random between 10 and 100 units. The third dataset is called *Random Dataset II* where 80 tasks were generated with deadlines uniformly random from zero to 1000, and the range of run times were generated uniformly random between 50 and 100 units. Obviously, Random Dataset II contains a much more restrictive set of tasks to schedule than Random Dataset I.

There are several issues to consider when implementing a MicroGA. Table I shows the effect of decreasing the population size for the Contrived Dataset of 80 tasks. As expected, as the population size decreases to a relatively small number, the quality of the solution decreases. To counter the negative effect of a small population size for a MicroGA, we experimented with increasing the mutation rate. Table II shows the results. Notice for our population size of 10, better results occur as the mutation rate increases from 10% to 100%. A 10% mutation rate means that 10% of the time a child chromosome will have a mutation operation performed on it. A 100% mutation rate means every child chromosome will have the mutation operation performed. In our case mutation consists of exchanging two randomly selected alleles in a chromosome. The results for the genetic algorithm entries shown in Table I through Table VI represent the average performance of executing the GA five times with different random number generator seeds. In Table III we compare FIFO, EDLF, and SRTF greedy algorithms and their associated genetic algorithms. We used the schedule obtained from FIFO, EDLF, and SRTF to seed an associated GA, called GA- FIFO, GA-EDLF, and GA-SRTF, respectively. For example, the SRTF solution was provided to the initial population of the GA-SRTF algorithm to allow the GA to improve upon it. Notice each GA was able to improve the results (in some cases significantly) over the initial schedule provided by the associated greedy algorithm. Recall that FIFO is simply the random order in which the tasks are arriving. Hence, GA-FIFO is the same as an unseeded GA. The GA results shown in Table III are for a population size 10, and a mutation rate of 100%, which is the model for our MicroGA. However, we allowed the number of trials to continue until convergence, rather than cutting it off at 10 trials as in our MicroGA. In this way we can see the long term positive effect of seeding a GA with one of the greedy heuristics.

IV. MicroGA Results and Conclusions

To simulate a real-time scheduling environment, our MicroGA uses the sliding window technique with a population size 10 and a mutation rate of 100%. We ran MicroGA on Random Dataset I and Random Dataset II and compared these results with EDLF, SRTF, and FIFO. The results are shown in Table IV. Considering both the number of completed tasks and the highest fitness value, the MicroGA performed slightly better than the EDLF algorithm for Random Dataset I. However, the MicroGA algorithm was clearly the superior algorithm using Random Dataset II. The reader should not be concerned with the low percentage of tasks completed in Random Data sets I and II. Since the tasks were generated randomly in each of the datasets, for all we know the results obtained in Table IV could be the optimal of near optimal scheduling for the tasks generated.

We extended our MicroGA to a parallel scheduling model to simulate a multi-processor environment. This simulation was run on a hypercube with one host and four processors. The general scheduler works on the host and its only function is to maintain a queue of tasks and distribute them to the working processors, as they become available. The working processors take the tasks from the host and process the tasks the same way that a sequential model would do. Each processor has a constant window of 10 tasks to analyze. Each of the four processors perform a MicroGA on the window set, execute the task determined by the MicroGA, then get another task from the host and start the process over again. In this model there is no communication between any of the four processors except with the host. However, a variation of this model allows tasks to migrate among the processors so that tasks falling behind in one processor can possibly be scheduled earlier on another processor. Hence we have a non-migration and migration model of a parallel task scheduler. Migration works as follows: The processors are connected in a ring topology with a left and right neighbor. At the conclusion of each MicroGA, each processor will schedule and run the task that appears as the first allele in the best chromosome in the population, just like the sequential MicroGA scheduler. If migration is to be performed, then a certain number of the largest delay tasks appearing in the best chromosome of each processor is passed to its left neighbor. We implemented this model by passing either 10% or 30% of the tasks at each step. For example, suppose tasks 13 and 56 were initially assigned to processor i, and are now going to be passed from processor i to its left neighbor. Suppose further that processor i receives tasks 24 and 41 from its right neighbor. Then for each chromosome in processor i, tasks (alleles) 13 and 56 are replaced by tasks 24 and 41, respectively. The sequential scheduling process then continues on each processor. We tested the parallel model using Random Data set II with no migration, 10% migration and 30% migration. Results are shown in Table V. Notice that migration definitely improves the parallel model significantly. Although it is not apparent if 10% or 30% is better. This research does suggest migration is better than no migration. We suggest 10% migration of the tasks over 30% migration to cut down on communication costs.

In summary, our results show that our MicroGA proved to be an excellent algorithm for task scheduling in a non-preemptive hard real-time scheduling environment. The steady-state model was determined to be better than the generational GA for our MicroGA. Our sequential MicroGA uses a small population size of 10 chromosomes, running for 10 trials using a mutation factor of 100% with a sliding window of 10 tasks. The parallel MicroGA works bests when migration is used to move tasks from one processor to another to even out the load as much a possible.

References

[1] John Biegel and James Davern, "Genetic Algorithms and Job Shop Scheduling", *Computers and Industrial Engineering*, 1990, Vol. 19, Numbers 1-4, pp. 81-91.

[2] J.L. Blanton and R.L. Wainwright, "Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms", *Proceedings of the Fifth International Conference on Genetic Algorithms*, July, 1993, Morgan Kaufmann, pp. 452-459.

[3] Ralph Burns, "Direct Chromosome Representation and Advanced Genetic Operators for Production Scheduling", *Proceedings of the Fifth International Conference on Genetic Algorithms*, July, 1993, Morgan Kaufmann, pp. 352-359.

[4] A.L. Corcoran and R.L. Wainwright, "A Genetic Algorithm for Packing in Three Dimensions", *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, 1992, pp. 1021-1030, ACM Press.

[5] A.L. Corcoran and R.L. Wainwright, "LibGA: A User-friendly Workbench for Order-based Genetic Algorithm Research", *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pp. 111-118, 1993, ACM Press.

[6] L. Davis, ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[7] K.A. De Jong and W.M. Spears, "Using Genetic Algorithms to Solve NP-Complete Problems", *Proceedings of the Third International Conference on Genetic Algorithms*, June, 1989, pp. 124-132.

[8] E. Falkenauer and S. Bouffouix, "A Genetic Algorithm for Job Shop", *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, April, 1991, Vol. 2, pp. 824-829.

[9] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W.H. Freeman, 1979.

[10] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[11] John Kanet and V. Sridharah, "PROGENITOR: A Genetic Algorithm for Production Scheduling", *Wirtscharfsinformatik*, August, 1991, Vol. 33, No. 4, pp. 332-336.

[12] Michelle Kidwell, "Using Genetic Algorithms to Schedule Distributed Tasks on a Bus-Based System", *Proceedings of the Fifth International Conference on Genetic Algorithms*, July, 1993, Morgan Kaufmann, pp. 368-374.

[13] Douglas Locke, "Scheduling in Real-Time", *Unix Review*, Vol. 8, No. 9, September, 1990.

[14] Nashat Mansour and Geoffrey Fox, "A Hybrid Genetic Algorithm for Task Allocation in Multicomputers", *Proceedings of the Fourth International Conference on Genetic Algorithms*, July, 1991, Morgan Kaufmann, pp. 446-473.

[15] P.P. Mutalik, L.R. Knight, J.L. Blanton and R.L. Wainwright, "Solving Combinatorial Optimization Problems Using Parallel Simulated Annealing and Parallel Genetic Algorithms", *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pp. 1031- 1038, 1992, ACM Press.

[16] G. Rawling, ed., *Foundations of Genetic Algorithms*, Morgan Kaufmann Publishers, 1991.

[17] Terry Shepard and J.A. Martin Gagne, "A Pre-run-time Scheduling Algorithm for Hard Real-Time Systems", *IEEE Transactions on Software Engineering*, Vol. 17, No. 7, July, 1991.

[18] David Tate and Alice Smith, "Expected Allele Coverage and the Role of Mutation in Genetic Algorithms", *Proceedings of the Fifth International Conference on Genetic Algorithms*, July, 1993, Morgan Kaufmann, pp. 31-37.

[19] D. Whitley, T. Starkweather, and D. Fuquat, "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator", *Proceedings of the Third International Conference on Genetic Algorithms*, June, 1989.

Population Size	Trial Converged	Tasks Completed	Fitness Value
100	2954	67/80	2777
50	657	63/80	440
20	47	54/80	-2513
10	16	50/80	-2865

Table I: Effect of Decreasing Population Size for the Contrived Dataset of 80 Tasks with Constant Mutation Rate = 10%

Population Size	Mutation Rate	Trial Converged	Tasks Completed	Fitness Value
10	10%	16	50/80	-2865
10	50%	76	58/80	-1265
10	100%	687	71/80	4545

Table II: Effect of Increasing Mutation Rate for the Contrived Dataset of 80 Tasks Using the Steady-State GA.

	Random Dataset I			Random Dataset II		
Algorithm	Trial Converged	Tasks Completed	Fitness Value	Trial Converged	Tasks Completed	Fitness Value
FIFO	–	46/80	-46,431	–	6/80	-203,785
GA-FIFO	1495	70/80	5,113	4799	15/80	-175,796
EDLF	–	69/80	6,085	–	0/80	-206,562
GA-EDLF	152	71/80	6,272	4257	15/80	-175,845
SRTF	–	54/80	-29,285	–	11/80	-178,845
GA-SRTF	1611	71/80	5,086	3734	16/80	-175,885

Table III: Comparison of Several Algorithms Using Random Dataset I and Dataset II. Steady-State GA was used with Population Size = 10, and Mutation Rate = 100%.

	Random Dataset I		Random Dataset II	
Algorithm	Tasks Completed	Fitness Value	Tasks Completed	Fitness Value
MicroGA	53/80	-21,233	10/80	-155,416
EDLF	53/80	-21,028	4/80	-204,733
SRTF	51/80	-33,034	7/80	-192,333
FIFO	46/80	-46,431	6/80	-203,785

Table IV: Real-Time Simulation Comparing MicroGA, EDLF, SRTF, and FIFO Using Random Dataset I and II. Population Size = 10, and Mutation Rate = 100%.

Test Runs	Migration Rate		
	None	10%	30%
1	39	43	50
2	39	47	47
3	42	40	44
4	36	42	43
5	36	60	50
AVE.	38.4	46.4	46.8

Table V: Parallel Real-Time Simulation Using four Processors. The Number of Completed Tasks is given using various Migration Rates. Dataset II was used.