GART: A Genetic Algorithm based Real-time System Scheduler

ManChon U*, Chiahsun Ho*, Shelby Funk, Khaled Rasheed
Department of Computer Science
The University of Georgia
Athens, GA 30602, USA
{manchonu,ho,shelby,khaled}@cs.uga.edu

Abstract—**Hard real-time systems require that all jobs are assigned a deadline and the system is deemed to be correct only if all jobs complete execution at or before their deadlines. Such strict timing requirements add to the complexity of the scheduling problem. This complexity is exacerbated when the system is executed on a multiprocessor platform. Even so, scheduling overheads must be kept to a minimum in order for the runtime behavior to be predictable. Thus, real-time scheduling algorithms have the dual requirement of satisfying complex requirements while using fairly simple and straightforward logic. One way an algorithm may achieve this goal is to reduce the overhead due to preemption and migration by rearranging the schedule so as to increase the duration between preemptions. Unfortunately, determining how best to rearrange the jobs is an NP-Complete problem. Hence, we need to use heuristics when scheduling such systems. This leads us to ask a couple of questions. First, what is the best heuristic? Second, is the same heuristic best for all real-time systems? This paper uses a Genetic Algorithm to help us answer these questions. Our genetic algorithm based real-time system scheduler (GART) is based on the DP-Wrap scheduling algorithm. The genetic algorithm searches through a variety of candidate heuristics to determine the best heuristic for a given task set. Experimental results demonstrate that this approach is able to efficiently identify the best heuristic for all the systems we consider. Moreover, we find that the "best" heuristic does, in fact, depend of various system parameters.**

Keywords- Genetic Algorithms; Real-Time System; Scheduling

I. Introduction

In hard real-time systems, the behavior of each job must be temporally correct in addition to being logically correct – i.e., not only do jobs have to perform the operations described by the code, these operations must be executed within a specific time frame. Such jobs are generally assigned a deadline. If a job does not complete before its deadline, it is considered to be a system failure. These systems are used whenever unpredictable behavior can lead to injury or loss of life (e.g., airplane autopilot system) or substantial financial losses (e.g., satellite controller). As one might expect, the critical constraints enforcing temporal correctness can cause a real-time scheduling algorithm to be quite complicated.

One aspect of real-time systems that can simplify the scheduling is the regularity of the workload. Real-time systems often execute jobs repeatedly at regular intervals. We call such a sequence of jobs a periodic task. If all jobs execute repeatedly in this manner, we say our system is comprised of a periodic task set. Several real-time scheduling algorithms are designed to schedule periodic task sets (e.g., [32, 33]). Like all schedulers, real-time scheduling algorithms must be very efficient – we want a system to be executing the real-time jobs as much as possible (rather than the scheduler). In particular, we wish to reduce the scheduling overhead the algorithm imposes through job preemptions. Unfortunately, minimizing the number of preemptions is an NP-Complete problem. If scheduling decisions are made online, we must use a heuristic to reduce the overhead. Moreover, this heuristic must be fairly simple so it can execute quickly. The goal of this research is to determine a method for finding such a heuristic for the multiprocessor scheduling algorithm DP-Wrap [16], a simple algorithm for scheduling periodic task sets on multiprocessor systems.

DP-Wrap is one of a number of schedulers that allocates processing time to tasks in proportion to each task's average demand. Such algorithms are said to adhere to the proportionate fairness principle. Any algorithm that adheres to this principle is optimal for multiprocessor systems. In 1996, Baruah, et al. [1], introduced P-FAIR, the first proportionate fair algorithm. By migrating tasks between processors, PFAIR can successfully schedule any task set whenever it is possible to do so. All of the fairness-based scheduling algorithms impose artificial deadlines in a manner that ensures all jobs share the same deadline, which greatly simplifies the scheduling decisions. Unfortunately, the simplicity of DP-Wrap comes at a cost – some jobs may be preempted frequently after executing only briefly. Hence, at the moment, DP-Wrap is of theoretical interest only – no system designer would choose to use DP-Wrap for their scheduler as long as causes so much overhead. We believe much of this overhead can be avoided.

Preemptions occur whenever a higher priority job arrives while a lower priority job is executing, forcing a context switch (i.e., the OS needs to store current-state information for the lower-priority task first, and load the higher priority task information into memory). Preemptions not only take time to implement, they also increase cache misses. Figure 1 illustrates a feasible schedule generated by original DP-Wrap and figure 2 illustrates a schedule generated by modified DP-Wrap (i.e., modified DP-Wrap uses a heuristic to choose which tasks to increase their execution time by utilizing available slack time). In Figure 1. , 7 preemptions occur on processor 1 (the number

*These two authors contributed equally to this work of context switches between task 1 and task 2). In contrast, there are only 2 preemptions in Figure 2. Hence, we were able to reduce this overhead by increasing the execution time of tasks during time slices. In summary, the modified DP-Wrap schedule (Figure 2.) reduces 88% context-switch overheads comparing to original DP-Wrap schedule (Figure 1.). This example demonstrates that violating the proportionate fairness rules may reduce context-switching overhead.

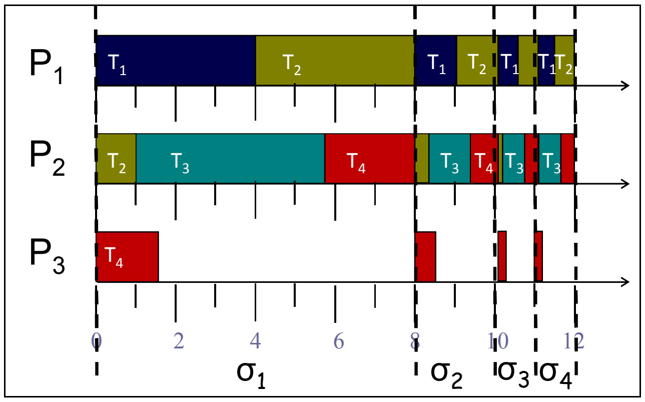


Figure 1. Original DP-Wrap Schedules

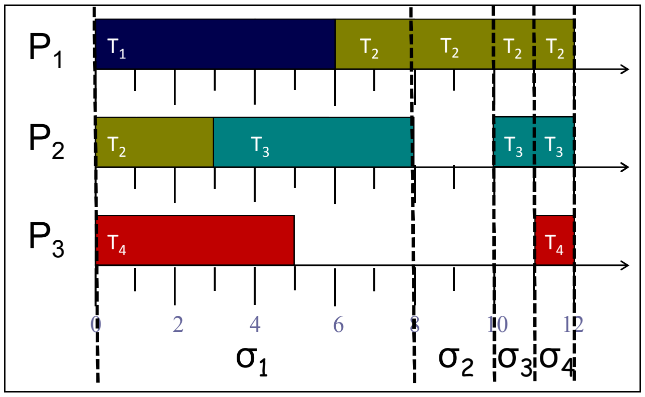


Figure 2. Modified DP-Wrap Schedules

This paper describes a method of using Evolutionary Computing techniques to determine which heuristic to use while scheduling a periodic task set on a multiprocessor platform. In particular, we consider a variety of ways to modify the DP-Wrap [16] algorithm to reduce the number of job preemptions. Employing a heuristic to modify when preemptions occur could make DP-Wrap a more practical option.

The goal of this paper is to determine heuristics for the DP-Wrap scheduling algorithm that reduce the number of preemptions and migrations of a given task set. In addition, some of these heuristics may be able to reduce the number of times the global scheduler must execute. Therefore, a well-selected heuristic can have a doubly good impact on the scheduling overhead. There are numerous choices for reasonable heuristics one might employ to achieve these goals. Furthermore, the "best" heuristic cannot be determined absolutely – the utility of a given heuristic will depend on a number of task set parameters. Exploring all possible heuristics for each task set would be unreasonably complex and time consuming. For this reason, we determined that Evolutionary Computation, specifically Genetic Algorithm-based Learning Classifier System approach, is the perfect tool for determining the best heuristic for a given task set. Our experimental results demonstrate that we are able to find the best heuristic fairly quickly using Pittsburgh approach classifier system techniques.

The rest of this article is organized as follows: Section II describes the background of this research work. Section III presents the details of our system implementation. The experimental results are presented in Section IV. Finally, we present the conclusion and future work in Section V.

II. Background

A. *Learning Classifier Systems*

Learning Classifier Systems (LCS) are evolutionary computation techniques which utilize GAs to discover new rules (or rule sets) in a problem domain. Traditionally there are two different approaches to construct LCSs: The Pittsburgh approach [12], and the Michigan approach [20]. In the Michigan approach, each individual is a rule and the full population constitutes a rule set while in the Pittsburgh approach each individual is a complete rule set (solution). Several variants of these two major types of LCS have been proposed in literature. XCS [13],[14] is the most well-known Michigan LCS. Wilson [15] investigated the use of XCS for data mining applications, and showed that XCS appears to have considerable potential for mining oblique datasets. E. Bernadó-Mansilla, et al. [16], compared the learning performance (prediction accuracy) of two genetic-based machine learning systems (GBML), XCS and GALE[25], with six well known learning algorithms, and it was shown that no method significantly outperformed both XCS and GALE. Saxon and Barry [17] applied XCS to the Monk's Problem, and demonstrated that XCS performs better than other machine learning techniques in some cases. On another note, the Pittsburgh approach gained more popularity lately. In [18], Llor`a and Garrell demonstrated how different knowledge representations can be coevolved in a fine-grained learning classifier scheme. Bacardit and Garrell[19] proposed a knowledge representation which evolves rules that can use multiple discretizations, letting the evolutionary process choose the correct discretization for each rule and attribute. Suraiya and Bharadwaj [26] used a Pittsburgh style LCS approach for automated discovery of Censored Production Rules. Furthermore, LCS applications can also be found in design problems and bioinformatics. In [21], L.I. Kuncheva and L.C. Jain suggested two simple ways to use a genetic algorithm (GA) to design a classifier fusion system, and tested their system with four real data sets. Bioinformatics-oriented Hierarchical Evolutionary Learning (BioHel) [23] applies the Iterative Rule Learning (IRL) approach to generate sets of rules. BioHel is inspired by GAssist[24], a Pittsburgh approach Learning Classifier System.

In GART, we have used the Pittsburgh approach to implement our own LCS, which means every single individual is a complete solution to the classification problem. We used a variable-length representation in which every individual is a variable-size set of rules. The details of our genetic algorithm will be presented in Section III.B.

B. Real-Time Task and Scheduling Model

In this section we describe our processing model. We also describe the DP-Wrap scheduling algorithm and the various heuristics we use to modify the algorithm. We begin by providing a more formal definition of the real-time task set.

1) Periodic Tasks

As mentioned above, real-time jobs may execute repeatedly at regular intervals. We call these repeating jobs periodic tasks. Each periodic task T_i is characterized by two parameters — a worst case execution requirement (WCET), e_i, and a period, p_i. We let $\tau \equiv \{T_1, T_2 ... T_n\}$ denote the real-time task system comprised of n periodic tasks. Each periodic task T_i generates an infinite sequence of jobs, $T_{i,0}, T_{i,1}, ..., T_{i,k}, \ldots$, which arrive exactly p_i time units apart. In this work, we assume an implicit task system, meaning each job's deadline is equal to the arrival time of the next job in the sequence. Thus, p_i denotes the length of time between each of $T_{i,k}$'s arrival and deadline and each job of T_i must be allowed to execute for e_i time units during the p_i time units after it arrives.

One useful parameter associated with each task set T_i is its utilization $u_i = e_i / p_i$, which is the proportion of time the task must be allowed to execute on average. The task set's total utilization, $U(\tau)$, is the sum of all the tasks' utilization values. A task set τ can be successfully scheduled on m processors if and only if $U(\tau) \leq m$ and $u_i \leq 1$.

2) The DP-Wrap Scheduling Algorithm

The DP-Wrap Scheduling Algorithm [11] is guaranteed to be optimal for scheduling periodic tasks on multi-processor platforms – i.e., DP-Wrap will schedule all tasks to meet all their deadlines if it is possible to do so. DP-Wrap divides the continuous time line into time slices. Each time slice's start time and finish time coincides with some job's deadline. At the beginning of each slice, all jobs are allocated a workload for the time slice and these workloads share the same deadline (namely, the end of the time slice). DP-Wrap allocates the workload of each task to be proportional to its utilization at the beginning of each time slice. In particular, if the length of the time slice is L then each task T_i will be allocated a workload of $u_i \times L$ at the beginning of the time slice. Because time slice boundaries coincide with task deadlines (and, hence, arrival times) it is easy to see that all jobs will meet their deadlines using this strategy. The heuristics we examine explore methods of initializing the workload to some other value without risking a deadline miss.

3) Time Slices(σ)

We let $t_0 = 0$ and t_1, t_2 ... denote the distinct deadlines of the tasks in τ, where $t_j < t_{j+1}$ for all $j \geq 0$. Then the j^{th} time slice, denoted σ_j, is $[t_{j-1}, t_j)$, and has length $L_j = t_j - t_{j-1}$.

4) Local Execution Remaining&Utilization

The local execution remaining of a task T_i at time t, denoted $l_{i,t}$, is the amount of time that T_i must execute before the next time slice boundary, i.e., between times t_{j-1} and t_j. As noted above, T_i's local execution is initialized to $u_i \times L_j$ at the beginning of each time slice. As T_i executes, $l_{i.t}$ decreases. In order for the algorithm to be correct, each task must have zero local utilization at the end of the time slice. A task's local utilization $r_{i,t} = l_{i,t} / (t_j - t)$ is the proportion of that time that T_i must execute during the remainder of the time slice. We let L_t and R_t denote a task set's summed local remaining execution and real-time respectively, at time t.

5) System Slack

We define the slack of a task set to be $S(\tau) = m - U(\tau) \times L$. For example, consider scheduling a task set with $U(\tau) = 1.5$ during a time slice of length 10 on2 processors. The processors have the capacity to do 20 units of work, but the tasks only have 15 units of work to be done. Therefore, 5 units of idle time must appear somewhere within the time slice. We call these extra units the system's slack within the current time slice. Part of our strategy for reducing overhead exploits the slack within each time slice.

III. System Details

A. DP-Wrap Simulator (DPWS)

The DP-Wrap scheduling algorithm calculates each slice boundary based on upcoming deadlines and allocates each task's workload by multiplying its utilization by the length of the time slice. Clearly, DP-Wrap can create a tremendous amount of preemption if time slice lengths are small. This is the reason that DP-Wrap is and impractical algorithm.

For above reason, we proposed a modified DP-Wrap scheduling algorithm which has two features: First, as Figure 3. describes, to select a task according to pre-given heuristic to increase its execution time by using the slack of time during each time slice. For example, if there are ξ idle time units during the current time slice and the pre-given heuristic is the earliest deadline first, DP-Wrap will select a task which has the earliest deadline then increase its execution time by up to ξ time units. Second, as stated in Figure 4. to select tasks according to one pre-given heuristic, to decrease its execution time, and then select another task, according to another pre-given heuristic, to increase its execution time. For instance, the increasing heuristic may be the earliest deadline first and the decreasing heuristic may be the least remaining execution time. DP-Wrap will select a task A which has the least remaining execution time for decreasing, and select a task B which has the earliest deadline for increasing – effectively allowing A and B to "swap" their execution times.

```
/* Utilize Available Slack Time */
while(slack > 0 ){
    while (eligible set is not empty) {
        find task (eligible set, heuristic increase);
        remove task from eligible set
        if (task increase amount < slack) {
            increase task (increase amount);
            slack -= increase amount;
        } else {
            increase task(slack);
            slack = 0;
        }
        if (task has not finished)
            add task back to eligible set
        }
}
```

Figure 3. Modified DP-Wrap Algorithm (Available Slack Time)

```
/* Utilized Ahead Execution Time */
while(eligible set && ahead set are not empty ){
    find increase task (eligible set, heuristic increase);
    find decrease task (ahead set, heuristic decrease);
    remove increase task from eligible set;
    remove decrease task from ahead set;
    determine ahead time (decrease task);
    increase amount = $min_{time}$(ahead, eligible increase);
    increase task (ahead amount);
    if (increase amount < ahead time) {
        ahead time -= increase amount;
        add decrease task back to ahead set;
    } else {
        add increase task back to eligible set;
    }
}
```

Figure 4. Modified DP-Wrap Algorithm (Ahead Execution Time)

In summary, the modified DP-Wrap has two strategies for changing task's local execution times: one for increasing execution times and another for decreasing them. We use eight different policies for selecting which tasks get increased time and which ones get decreased time, namely, Earliest Deadline First, Latest Deadline First, Least Remaining Execution Time First, Most Remaining Execution Time First, Least Laxity First, Most Laxity First, Smallest Period First, and Largest Period First. Our purpose is to reduce overheads by combining different heuristics.

B. Genetic Algorithm in GART

In this section, we describe the implementation details of the evolutionary component in our system. We used the Watchmarker Framework [2] to implement our evolutionary component since it is a highly extensible framework that gives us the flexibility to implement our own evolutionary algorithm in a fast and sophisticated way. Both a Generational Genetic Algorithm (GGA) and a Steady State Genetic Algorithm (SSGA) were implemented in our GART.

1) Individual (Chromosome)

In GART, we have originally implemented our individuals with fixed length (static) representation, yet later on we found out that some of the sub-chromosomes within the individual were useless. We therefore switched to variable length (dynamic) representation. Each individual contains from 9 to 65 variable sub-chromosomes with the last being a default sub-chromosome (described below). Each sub-chromosome consists of 14 genes which are the lower bounds and upper bounds of 7 main parameters of the system. Variable sub-chromosomes are the ones that participate in the evolution by the GA, while the default sub-chromosome consists of fixed values for the first 6 pairs (1st – 12thallele) of genes that represent the maximal range of the first 6 main parameters, as well as variable H_a (13th allele) and H_m (14th allele). One reason for using individuals with variable length is that an individual with more sub-chromosomes might not perform better than one with fewer sub-chromosomes, and vice versa. Furthermore, the optimal number of sub-chromosomes in an individual is not easy to determine. Therefore we put the length of an individual (number of sub-chromosomes) into the evolutionary process; we let the GA choose how many sub-chromosomes the best individual should contain. The aforementioned 7 main parameters of the system are listed as follows:

- Number of processors ($\boldsymbol{m}$)
  - Possible value: 2, 4, 8, 16, 32
- Minimum utilization ($\boldsymbol{U_{min}}$)
  - The minimum utilization among entire task set
  - Possible value: 1% to 30%
- Maximum utilization ($\boldsymbol{U_{max}}$)
  - The maximum utilization among entire task set
  - Possible value: 10% to 95%
- Total utilization divided by processor number ($\boldsymbol{U(\tau)/m}$)
  - Possible value: 30% to 95%
- Max utilization dived by min utilization ($\boldsymbol{U_{max} / U_{min}}$)
  - Possible value: 2 to 100
- Number of task dived by number of processors ($\boldsymbol{n/m}$)
  - Possible value: 2 to 25
- Heuristic
  - Addition Heuristic ($\boldsymbol{H_a}$), Minus Heuristic ($\boldsymbol{H_m}$)
  - Possible value:0 to 8

2) Initialization

```
/* Generate gene */
geneGenerator(){
for 1 to NumOfSubChromosomes{
    for each pair ofmain parameters{
        min = lower bound of the main parameter;
        max = upper bound of the main parameter;
        min'= getScaledGene(min, max);
        max'= getScaledGene(min', max);
        individual.add(min');
        individual.add(max');
    }
    return individual;
}
/* Get scaled gene */
getScaledGene(min, max){
   if (input is integer)
       return (this.rand.nextInt(max-min) + min);
   else
       return (this.rand.nextDouble() * (max-min) + min);
}
```

Figure 5. Modified Function of Generating Random Individuals

The initial population of our GA consists of 200 individuals with various randomly generated gene values. All of these 200 different individuals are generated by a uniformly random function that takes into account the lower bound and upper bound of each main parameter of the sub-chromosome as constraints. The function for generating each individual is described in Figure 5.

3) Parent Selection

We used the deterministic binary tournament selection operator [27] for parent selection. The operator picks a pair of candidates at random and then selects the fitter of the two candidates to become a parent and/or join the mating pool.

4) *Evolutionary Operators*

a) *Crossover*

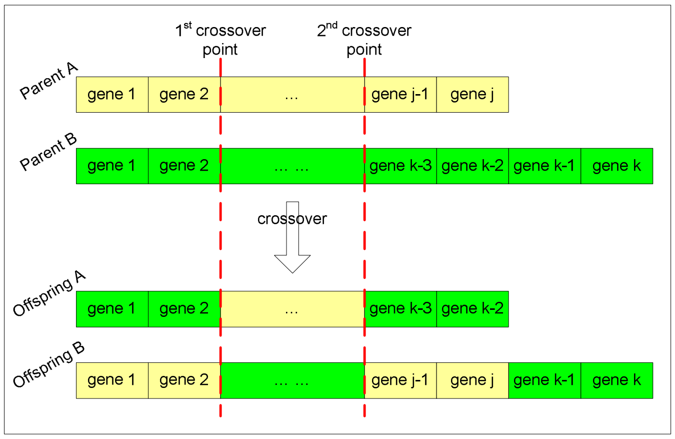


Figure 6. Crossover between Individuals in Different Length

In order to preserve the simplicity and the efficiency of the system, we used two-point crossover in our evolutionary component. It works by choosing two random numbers r_1 and r_2 in the range of [1, l-1] (where l=the length of the shorter parent), and then splitting the selected parents at r_1 and r_2 into three segments of contiguous genes, and then the offspring are created by taking alternative segments from the two parents. If the two parents have different length, then l will be the length of the shorter parent in an effort to prevent bloating as Figure 6. illustrates.

b) *Mutation*

We have used non-uniform mutation [27] in our GA with probability 5%, which means each selected individual has 5% probability to be mutated, and the mutation applies to every single gene inside the selected individual. The mutation amplitude shrinks as the generation number t increases. As a result, this operator searches the space uniformly initially (when t is small), and very conservatively at later stages.

c) *Repair Operator*

Since the possibility of having a greater min value than the max value may occur after the crossover and mutation, we have implemented a repair operator to ensure that the value of a lower bound gene should be smaller than the corresponding upper bound gene value. The operator is invoked if the min value is greater than the max value for some range. We first generate a uniformly random binary number. If the random binary is zero then we replace our min value gene with a random value from the lower bound to max value minus one, otherwise we replace the max value gene with a random value from min value plus one to the upper bound.

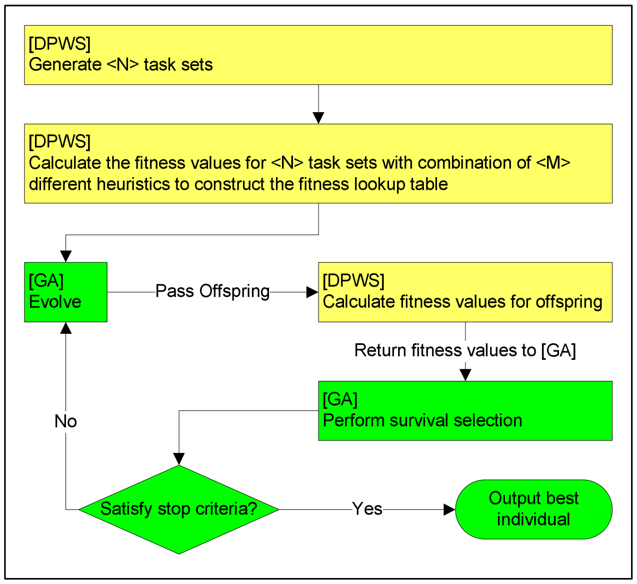
5) *Fitness Evaluation*

In GART, the fitness value of an individual is its simulation result returned from the DPWS. Whenever the DPWS receives an individual from the GA, it begins the simulation for scheduling the 1,000 task sets with the rules inside the individual. Here, every sub-chromosome of the individual is a unique rule without any duplication with other sub-chromosomes in the same individual. After the simulation, DPWS returns the fitness (simulation result), which is the total amount of preemption, migration, and wakeup numbers, to the GA. The smaller fitness value DPWS returns, the better the individual is.

6) *Survivor Selection & Elitism*

Sometimes good candidates can be lost when crossover or mutation results in offspring that are weaker than the parents. Often the EA will re-discover these lost improvements in a subsequent generation but there is no guarantee. To combat this we use a feature known as elitism. Elitism involves copying a small number of the fittest candidates, unchanged, into the next generation. This can sometimes have a dramatic impact on performance by ensuring that the EA does not waste time re-discovering previously discarded partial solutions. Candidate solutions that are preserved unchanged through elitism remain eligible for selection as parents when breeding the remainder of the next generation. Therefore, we set the elitism to 1.

C. *System Workflow*



[DPWS]
Generate <N> task sets
[DPWS]
Calculate the fitness values for <N> task sets with combination of <M> different heuristics to construct the fitness lookup table
[GA]
Evolve
Pass Offspring
[DPWS]
Calculate fitness values for offspring
Return fitness values to [GA]
[GA]
Perform survival selection
No
Satisfy stop criteria?
Yes
Output best individual

Figure 7. GART System Flow Chart

As Figure 7. illustrates, the general workflow of our GART system can be summarized as below:

Step 1. DPWS randomly generates 1,000 task sets

Step 2. DPWS Calculate the fitness vales for the 1,000 task sets with different combination of our 8 different heuristics for constructing the fitness lookup table

Step 3. GA read in the 1,000 task sets as initial population, then start evolving

- Step a. GA generate offspring by crossover, and mutation
- Step b. GA passes the new generated offspring to DPWS for fitness calculation
- Step c. DPWS returns the fitness values to GA
- Step d. GA performs survivor selection

Step e. If stop criteria is met then go to Step 4, otherwise go to Step a

Step 4. Output best individual (Final classification result)

IV. EXPERIMENTAL RESULTS

A. System Enviornment

1) Platform

- Processor: Intel(R) Xeon(R) CPU X3430 @2.40GHz
- Installed Memory (RAM): 4.00 GB
- Operating System: Linux version 2.6.18-194.26.1.el5, Red Hat 4.1.2-48
- Java Version: Java(TM) SE Runtime Environment: 1.6.0 (build pxi3260sr9-20101125_01(SR9))
- Genetic Algorithm Framework: Watchmaker [2]v0.7.1

2) Task Sets

We applied our approach to multiprocessor systems, with the number of processors ranging from 2 to 32 (i.e., 2, 4, 8, 16, and 32). We used Baker's method to randomly generating task sets [3], we generated approximately 10,000 task sets with a variety of different characteristics. Each task set was generated for a target number of processors. In addition, each task set had a target maximum utilization U_{max}, which ranged from 0.1 to 0.9. For each randomly generated task set, task periods were selected between 10 and 200, and tasks' utilizations were randomly selected in the range (0, U_{max}]. We use the period and the utilization to determine the task's execution time. The task sets were generated with a target total utilization ranging from 0.25% to 97.5% of *m*. For instance, if we were generating a task set for 4 processors with a target total utilization of 80%, then our target total utilization would be $U(\tau) = 3.2$. Total utilization of a task equals to its execution divided by its period (utilization = exec/ period), thus the total utilization equals to:

$$\text{Utilization}_{\text{Total}} = \sum_{\text{i=1}}^{\text{All tasks in one task set}} \frac{\text{exec}^{\text{i}}}{\text{period}^{\text{i}}}$$

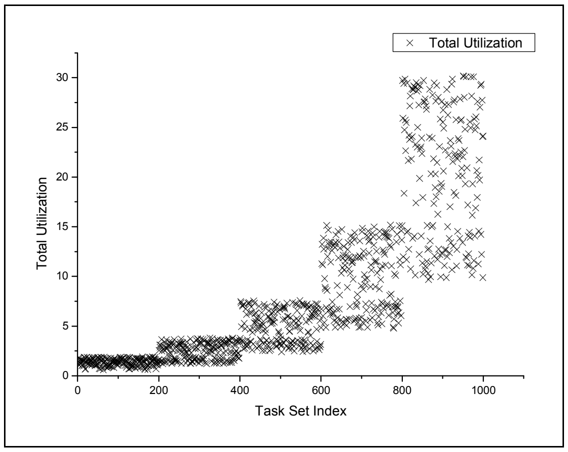


Figure 8. Histogram of Total Utilization of All Task Sets

Figure 8. shows the histogram of all 1,000 task sets with different total utilization (2^i, i=1,2,3,4,5).

3) Genetic Algorithm

For your ease of reference, we have summarized the settings of our Genetic Algorithm in TABLE I.

TABLE I. DETAILED SETTINGS OF OUR GENERATIONAL GA (GGA) AND STEADY STATE GA (SSGA)

	GGA	SSGA
Gene Type	Double (Real Numbers)	
Sub-Chromosome Size	14 Genes	
Individual Size	9 to 65 Sub-Chromosomes	
Initial Population	200	
Crossover	2 Points	
Crossover Rate	100%	
Mutation	Non-Uniform	
Mutation Rate	5%	
Parent Selection	Tournament Selection	
Survivor Selection	N/A	Tournament
Selection Size	3	
Elitism	1	
Max Generation	3000	N/A
Max Fitness Evaluation	N/A	1000*200

B. Result Analysis

As we stated in Section III.B, we have used the Pittsburgh approach to implement our own LCS in GART, which means every single individual is a complete solution to the classification problem, and every individual is a variable length set of rules, which contains *n* sub-chromosomes where *n* is randomly chosen in the range from 9 to 65. Each sub-chromosome has 7 pairs of main parameters that are represented by 14 genes. Every single task set has its own values for the first 12 genes, which are the ranges of the first 6 of the main parameters. While the scheduling is being performed, GART classifies each task based on its corresponding values of aforementioned 12 genes: if sub-chromosome *i* (where $0<i<n$) do not fit, GART will move to check sub-chromosome *i+1*, and so on. If any of the first *n-1* sub-chromosomes matches with the task's corresponding values, then the task will be scheduled with the corresponding H_a and H_m that are located in the 13th and 14th genes of that sub-chromosome. Yet if all of the first *n*-1 sub-chromosomes have been checked and none of them matched with the task's corresponding values, then this task will be classified according to the *n*th sub-chromosome (the last one) as it is the default sub-chromosome which possesses the largest possible intervals of values of the first 6 (fixed) pairs of main parameters, and the 13th and 14th allele of it are the heuristics (H_a and H_m) selected by the GA.

In order to accurately evaluate the performance of GART during experiments, we let the DPWS generate one single set of task sets, which includes 1,000 task sets, as input. Then we have GART run 32 times independently from different random initial populations with this input and the fittest individual (best rule) is recorded at the end of every complete run.

Our records (in Appendix) indicate that GART always performed better than the minimum overhead generated by the single combination of various scheduling algorithms (Base Line) in all of the runs, with both GGA and SSGA. Furthermore, this always happens within 25 generations with GGA, and fewer than 5,000 fitness evaluations with SSGA. These two figures also provide further evidence to support that GART could obtain smaller overhead (smaller fitness value) given more generations. More detailed information of the 32 independent runs is listed in TABLE II. where the First and third rows present the average best-found fitness values of all complete runs, while the second and fourth rows describe their improvement relative to the fitness value generated by the original default heuristic ($H_a = 0$, $H_m = 0$), which is 18,200,000. The improvement generated by the global optimum found through exhaustive search is 67.1667% (fitness value = 5975664), which is very close to the average improvement of GART with GGA (67.0765%) and SSGA (67.0703%).

TABLE II. RESULTS OF 32 COMPLETE RUNS WITH GGA AND SSGA

	Avg. of 32 Runs	**Stdev.**
GGA Best Fitness	5992070.09	441.96
GGA Improvement	67.0765%	0.0024%
SSGA Best Fitness	5993210.59	556.32
SSGA Improvement	67.0703%	0.0031%

Note that with our GGA, every generation contains 200 fitness evaluations, while the SSGA contains only one for every evolution. Throughout the course of the experiments, we also noticed that the GGA performs better than the SSGA: since all of the 32 independent runs in GGA can obtain better fitness values within 8 to 25 generations (≈ 1,600 to 5,000 fitness evaluations) than the ones generated by the best single combination of Ha & Hm, but the SSGA needs around 2,700 to 13,500 fitness evaluations to fulfill this task. Figure 9. provides further evidence to this observation. The figure plots on the X-axis the first time a fitness value better than the baseline is found by each of the 32 runs, while the Y-axis plots the value of such fitness. You can see that all GGA runs reached the aforementioned fitness level within 5 minutes while some of the SSGA runs require up to 9 minutes.

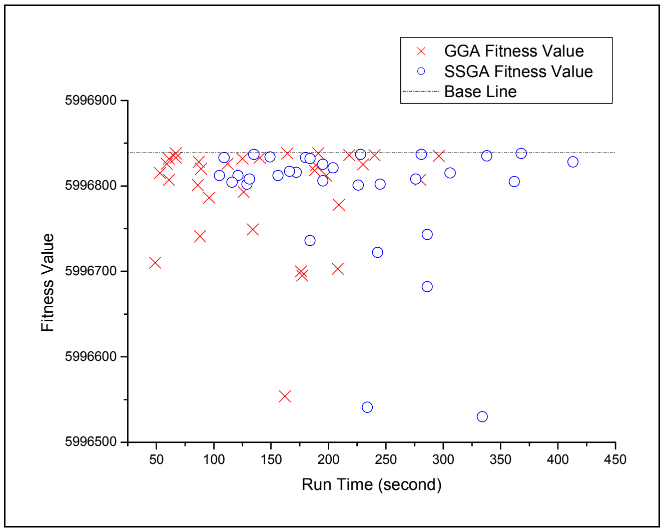


Figure 9. Run Time Comparison between GGA and SSGA with H_a & H_m

According to our observations, in the first generation, all task sets are classified to the n^{th} sub-chromosome as it possesses the largest intervals of values of the first 12 genes. And the total amount (fitness) of preemption, migration and wakeup numbers of the 1,000 task sets is the maximum overhead during the run. Yet after a few generations, the GGA can successfully reproduce much better individuals, and the GGA could find the individual that generates the minimum overhead as the one generated by the single combination of various scheduling algorithms, in less than 25 generations. Similar success is obtained with our SSGA as well.

Later on, we added another additional heuristic H_{a1} into our sub-chromosomes (without having lookup table implemented) in order to increase the complexity of the problem domain. This modification would lead to a dimensional increment for every single sub-chromosome making exhaustive search impractical. We execute the experiment for 32 separate runs. Based on Figure 10. it is not difficult to see that GART works even faster than previously (only with H_a and H_m), requiring fewer fitness evaluations (or generations) to perform better than the minimum overhead generated by the single combination of various scheduling algorithms (Base Line). However, this would require days to achieve by exhaustive search.

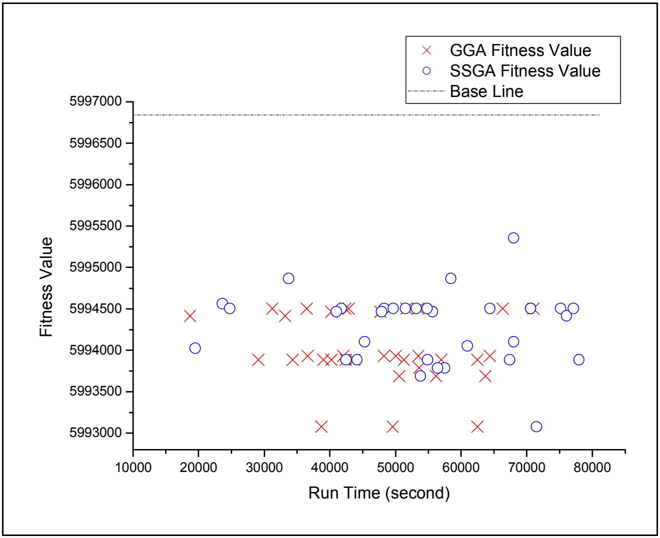


Figure 10. Fitness Values Comparison between GGA and SSGA with H_a& H_{a1}& H_m

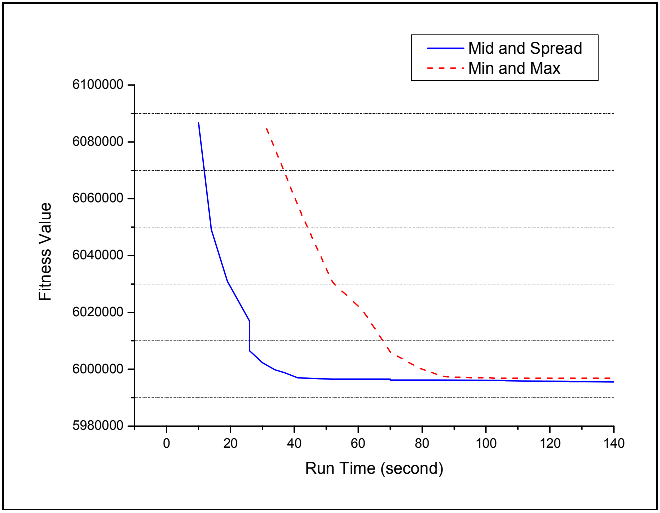


Figure 11. GGA with Mid & Spread and Min & Max Intervals Encoding

Furthermore, we experimented with a different method for encoding intervals, using the Midpoint and Spread instead of the Min and Max to specify intervals. The results are presented in Figure 11. which illustrates the average run time and the fitness values of the first 140 seconds of 32 independent runs that generated GART with the GGA kernel. The solid blue line is the implementation with Midpoint and Spread intervals encoding method, while the dotted red line is the implementation with Mix and Max intervals encoding method. The result indicates that with Midpoint and Spread interval encoding method, GART converges faster than with the traditional Min and Max encoding method. The final fitness values were not significantly different though. We were unable to present more comparison results due to space limitations, but we believe that the experimental results presented in this article provide ample evidence to the effectiveness and efficiency of GART, a novel genetic algorithm based real time scheduler.

V. Conclusions & Future Work

This article presents a genetic algorithm-based real-time system scheduler (GART), which determines the best heuristic for reducing scheduling overhead. The algorithm is based on the DP-Wrap scheduling algorithm, which is known to be optimal but to suffer from high overhead costs. Given that the problem of minimizing preemptions is NP-complete, we cannot expect to be able to determine the best strategy online. Instead GART selects the best heuristic for us. Our experiments show that GART successfully determines the best heuristic for all the systems we tested within reasonable time. This work is significant since it utilizes an evolutionary computation technique to make use of the different existing scheduling algorithms to construct a powerful scheduler, without involving profound research in the theory of scheduling.

In the future, we would like to implement other crossover and mutation techniques to see how the performance varies. Furthermore, we would also like to consider other heuristics and scheduling algorithms in our analysis. It is not uncommon for optimal scheduling algorithms to suffer from high overhead. Similarly, low-overhead algorithms may require long durations of idle time in order to ensure all jobs meet their deadlines. Applying the GART approach allows real-time scheduling algorithms to have the best of both worlds – lower overhead with high processor utilization – whenever possible.

VI. Appendix

Details of our experiments presented in this article are available at http://ecml.uga.edu/gart_cec_2011_sup.xlsx

References

[1] S. K. Baruah and J. Carpenter. Multiprocessor Fixed-Priority Scheduling with Restricted Interprocessor Migrations. Journal of Embedded Computing, 1(2):169–178, 2004.

[2] Watchmarker Framework – available at http://watchmaker.uncommons.org/

[3] T. P. Baker, "An analysis of fixed-priority schedulability on a multiprocessor," IEEE Real-Time Systems Symposium (RTSS), vol. 32, no. 1-2, pp. 49–71, 2006.

[4] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," IEEE Real-Time Systems Symposium (RTSS), vol. 0,pp. 387–397, 2009.

[5] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," Journal of the ACM, vol. 20, no. 1, pp. 46–61, 1973.

[6] M. Dertouzos and A. K. Mok, "Multiprocessor scheduling in a hard real-time environment," IEEE Transactions on Software Engineering, vol. 15, no. 12, pp. 1497–1506, 1989.

[7] M. Dertouzos, "Control robotics : the procedural control of physical processors," in Proceedings of the IFIP Congress, pp. 807–813, 1974.

[8] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. IEEE Embedded and Real-Time computing Systems and Applications (RTCSA), 2006

[9] H. Cho, B. Ravindran, and E. Jensen. An Optimal Real- Time Scheduling Algorithm for Multiprocessors. International Real-Time Systems Symposium (RTSS), 2006.

[10] D. Zhu, D. Moss´e, and R. Melhem. Multiple-Resource Periodic Scheduling Problem: how much fairness in necessary? International Real-Time Systems Symposium (RTSS), 2003.

[11] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in Euromicro Conference on Real-Time Systems (ECRTS), pp. 1–10, 2010.

[12] S. F. Smith. A Learning System Based on Genetic Adaptive Algorithms. PhD thesis, The University of Pittsburgh, 1980.

[13] Wilson, S. W.,"Classifier Fitness Based on Accuracy," Evolutionary Computation, 3(2):149–175, 1995.

[14] Wilson, S. W., "Get Real! XCS with Continuous-valued Inputs," In Lanzi, P.-L., Stolzmann, W., and Wilson, S. W., (Eds.), LNAI 1813 : From Foundations to Applications, pages 209–220. Springer Verlag, 2000.

[15] Wilson, S. W., "Mining Oblique Data with XCS," IlliGAL Report No. 2000028, 2000.

[16] E. Bernadó-Mansilla, X. Llorà, and J.M. Garrell-Guiu, "XCS and GALE: A Comparative Study of Two Learning Classifier Systems with Six Other Learning Algorithms on Classification Tasks," Proc. Fourth Int'l Workshop Learning Classifier Systems (IWLCS '01), short version published in Proc. Genetic and Evolutionary Computation Conf. (GECCO '01), pp. 337-341, 2001.

[17] Saxon, S. & Barry, A.,"XCS and the Monk's Problems," In P-L. Lanzi, W. Stolzmann & S.W. Wilson (eds) Learning Classifier Systems: From Foundations to Applications. pp 223-242. Springer, 2000.

[18] Llor`a, X. and Garrell, J. M., "Co-evolving different knowledge representations with fine-grained parallel learning classifier systems," Proceeding of the Genetic and Evolutionary Computation Conference (GECCO2002), 2002

[19] Bacardit, J. and Garrell, J. M., "Evolving multiple discretizations with adaptive intervals for a Pittsburgh rule-based learning classifier system," Genetic and Evolutionary Computation Conference (GECCO2003), pp 1818–1831, Berlin. Springer-Verlag, 2003.

[20] Holland, J. H. and Reitman, J. S., "Cognitive Systems based on Adaptive Algorithms," Pattern Directed Inference Systems, 7(2):125–149, 1978.

[21] L.I. Kuncheva and L.C. Jain, Designing classifier fusion systems by genetic algorithms, IEEE Trans. Evol. Comput. 4, pp. 327–336, 2000.

[22] Zbigniew Michalewicz, Genetic algorithms + data structures = evolution programs, Springer-Verlag, 1996.

[23] Bacardit J, Krasnogor N, "Biohel: Bioinformatics-oriented hierarchical evolutionary learning". Nottingham eprints, University of Nottingham, 2006.

[24] Bacardit J, "Pittsburgh genetics-based machine learning in the data mining era: Representations, generalization, and run-time," PhD thesis, Ramon Llull University, Barcelona, Spain, 2004.

[25] DeJong, K.A., Spears, W.M.,"Learning concept classification rules using genetic algorithms," Proceedings of the International Joint Conference on Artificial Intelligence, pp: 651–656, 1991.

[26] Jabin, Suraiya and K. K. Bharadwaj, "Learning Classifier SystemsApproach for Automated Discovery of Censored Production Rules," In 14. International Enformatika Conference, v14-57, Vol 14, 2006.

[27] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," Journalof the ACM, vol. 20, no. 1, pp. 46–61, 1973.

[28] M. Dertouzos and A. K. Mok, "Multiprocessor scheduling in a hard real-time environment," IEEE Transactions on Software Engineering, vol. 15, no. 12, pp. 1497–1506, 1989.