akistan Journal of Information and Technology 1(2): 153-159, 2002

Scheduling Real-Time Tasks in Distributed Environment with Tabu Search Algorithm

A. Mahmood
Department of Computer Science, University of Bahrain, Bahrain

Abstract: Many time-critical applicaticns require predictable performance and tasks in these applications have to meet their deadlines. Hence, tasks in these applications need to be scheduled in such a manner that they not only meet their deadlines but also satisfy some performance criteia specific to the application doamian. Scheduling real-time tasks with minimum jitter in a distributed computing environment is particularly important in many control applications. This problem is known to be NP-hard, even for simple cases. Therefore, heuristic approaches seem appropriate to these classes of problems. In this paper, we investigate a tabu search algorithm for nonpreemptive static scheduling of real-time tasks where tasks are periodic and have arbitrary deadlines, precedence, and exclusion constraints. The proposed algorithm not only creates a feasible schedule but it also minimizes jitter for periodic tasks. The performance of the algorithm has been studied through a simulation and the results are reported in this paper.

Key Words: Distributed Systems, Task Allocation, Tabu Search, Real-Time Systems, Task Scheduling, Jitter

Introduction

Real-time systems are characterized by computational activities with timing constraints (deadlines) that have to be met in order to achieve desired behaviour. Some of the tasks are periodic in nature and need to be cyclically executed at constant activiation rates. Other tasks are aperiodic and are activated only upon the occurance of a particular event. Hence a periodic task consists of a sequence of identical jobs that are regularly activated at a constant rate.

Many tasks performed in systems such as process control, air defense, and space exploration are inherently distributed and have real-time constraints (Stankovic *et al.*, 1985). In those cases where the environment and the application characteristics are known a *priori*, the worst-case conditions and critical rates can be evaluated (Natal *et al.*, 2000). The computation required by the applications can be distributed in a set of periodically activated tasks being scheduled according to a fixed pattern. The scheduler is run off-line and uses the parameters of the task set to generate a table of activation times to be used by the local dispatchers.

In some applications, feasible scheduling of all the task instances within the deadline is not sufficient to generate the correct behavior of the system. Two instances of the same periodic task may have to be separated by an amount of time variable between zero and the two periods minus the computation time (Natal *et al.*, 2000). In some cases, this variation is a serious problem and there is a need for different scheduling to minimize jitter. For example, the Airplane Information Management System of the Boeing 777 required the minimum jitter on the order of 100 *μs* and 1 *ms* (Carpenter *et al.*, 1994). The kind of jitter described in (Carpenter *et al.*, 1994) is the problem we solve in this paper.

Another case study on the development of the Olympus Satellite Attitude and Orbital Control System (Burns *et al.*, 1993) cites the minimum output jitter as one of the requirements in task scheduling. A maximum jitter of 5 *ms* is given as the requirement for a fixed priority scheduler in yet another aircraft application (Bate *et al.*, 1996). Even though the output jitter is often considered as a part of the constraints, the performance of digital control system actually depends on the amount of jitter. In these cases, a minimum jitter is preferable (Natal *et al.*, 2000).

Most of the previous research on task scheduling has been restricted to solve the feasibility problem only (i.e. schedule tasks within their deadline). Most of the proposed solutions techniques use heuristic-driven search algorithms to find the feasible task sequence. Branch-and-bound algorithm by Xu and Parnas (1990) schedules tasks with exclusive relationship and precedence constraints. Mars Scheduler (Fohler 1994) uses an iterative deepening search algorithm to solve the distributed scheduling problem. For multiprocessor systems with resource-constrained tasks, a heuristic search algorithm, called myopic scheduling algorithm, was proposed in (Ramamritham *et al.*, 1990). It was shown by the authors that an integrated heuristic which is a function of deadline and earliest start time of a task performs better than simple heuristics, such as EDF, least laxity first (i.e. a task with least value of deadline minus computation time executes first), and minimum processing time first. The myopic algorithm was latter improved by (Manimaram and Murthy, 1998) by introducing the concept of feasibility check window and parallelizing a task whenever deadline of a task cannot be met. It was shown through an intensive simulation study that the improved myopic algorithm outperforms the original algorithm. If we try to use these traditional search-based solutions, we need a fairly complex heuristic function to achieve the goal of minimizing jitter. To the best of our knowledge, no such heuristic function has been proposed.

The genetic algorithm community has also proposed some genetic algorithms for task scheduling problem. Kim and Hong (1993) proposed a genetic algorithm for static scheduling of tasks with the objective of minimizing total communication costs while achieving a load balance. Hou *et al* (1994) presented a genetic algorithm for static scheduling of non-identical tasks with known execution time and precedence relationships to identical processors. Their goal was to minimize the finishing time of the schedule. Kidwell and Cook (1994) used a genetic dynamic scheduling algorithm to assign non-identical tasks to identical processors. Recently, Mahmood (2000) proposed a hybrid genetic algorithm to schedule hard real-time tasks on multiprocessor systems.

Tabu search metaheuristic has also been applied to task allocation and scheduling problems. Porto and Ribeiro (1995) applied a tabu search metaheuristic to the solution of task scheduling problem on a heterogeneous multiprocessor environment under precedence constraints. Each task is assigned to a particular processor through an allocation function. A neighborhood solution is obtained by taking a single task from the task list of a processor and transferring it to a different processor. The whole neighborhood is obtained by going through every task and then building a new solution by placing the task into every position of the task list of every other processor in the system. A *recency-based* tabu mechanism was used to make certain moves tabu for a number of iterations. In (Hubscher and Glover, 1996), a tabu search algorithm was proposed to minimizing the makespan on *n* tasks on *m* equivalent processors. The authors used a candidate list strategy that generates only a small subset of moves. A dynamic tabu list is employed for handling tabu restrictions. They also used an influential diversification to improve the quality of the solution. Mahmood (2002) recently proposed a tabu search algorithm for scheduling real-time tasks under precedence and resource constraints.

Simulating annealing techniques have also been used by Tindell *et al* (1992) to find optimal processor binding for real-timer tasks to be scheduled according to fixed priority policies. Natale and Stankovic (2000) applied simulating annealing technique for scheduling real-time tasks having arbitrary deadlines, precedence and exclusion constraints. Their solution not only creates feasible schedules, but also minimizes jitter for periodic tasks.

In this paper, we propose the application of tabu search technique to static scheduling of real-time tasks in a distributed computing environment. The algorithm not only creates a feasible schedule, but it also minimizes jitter for periodic tasks. To the best of our knowledge, tabu search has not been applied to this problem before. Note that, this paper uses, rather than extends, the tabu search technique. This paper, however, does provide a new application of tabu search to an important problem. It also provides a transformation of this problem into a form that is amenable to the application of tabu search. We also demonstrate, through a simulation study, the performance and value of the algorithm on experimental task sets for different loads and attributes.

The rest of the paper is organized as follows. Next section presents system model and problem formulation. The problem has been formulated as a combinatorial optimization problem. Basic tabu search algorithm is presented in the next section followed by a description of how the basic tabu search algorithm can be specialized into a specific algorithm for task scheduling. This is followed by simulation results. The paper is concluded in the last section.

System Model and Problem Formulation: We consider a set of *n* periodic processes $P_1, P_2 \cdots P_n$ statically assigned to a set of processors connected through a shared medium. Each process instance p_i in the *scheduling period*, defined as the least common multiple of the processes' periods, is characterized by a deadline d_i, a release time r_i, an activation period P_i and CPU specification to which it is statistically assigned.

The scheduling period is *lcm(P)*. Each process is divided into a chain of scheduling units, the *tasks*. The tasks can be defined as sequential nonpreemptive computational units that begin and end with a synchronization point or with a basic operation (request or release) on one or more resources. Each time a process instance is executed, the entire set of tasks for this process must execute.

The tasks inherent the time attributes of the processes to which they belong to and have other additional attributes of their own. Each task instance t_i in the scheduling period is characterized by a deadline $d_{i'}$, a computational time $c_{i'}$, a set of resources requested in exclusive mode $\{R_{t_i1}, R_{t_i2} \cdots R_{t_in}\}$ and a set of precedence constraints $\{t_i \to t_j, t_i \to t_k \cdots t_i \to t_m\}$ meaning that task t_i must be completed before the execution of tasks $\{t_j, t_k \cdots t_m\}$.

We define the *starting time* s_{ik} as the time instance when t_{ik}, the *k*th instance of task t_i (where $k = 1,2,\cdots m_i$) is successfully scheduled and task

```
abu_search(instance_list, maxmoves, neigh_size)

    initialize the short-term memory
    generate a starting solution $s_0$
    $s, s^* = s_0$  /* s is curent solution and s* is best solution
                   found so far */
    for(i=1; i<=maxtry; i++) {
          bestmovevalue=∞
          for (all candidate moves in the neighbourhood)
              if ( the candidate move is admissible) {
                    obtain the neighbor solution $\overline{s}$ by applying a candidate move to the current solution s
                    movevalue= $c(\overline{s}) - c(s)$ /* c(s)is the objective function
                                                   to me optimized */
                    if (movevalue<bestmovevalue){
                          bestmovevalue=movevalue
                          $s' = \overline{s}$
                    }
              }
    }
    update the short term memory function
    if $(c(s') < c(s^*))$ $s^* = s'$
    $s = s'$
}
```

Fig. 1: A General Framework of Tabu Search

```
tabu_search(instance_list,maxmoves,neigh_size)
{
  init_schedule(instance_list); //find initail schedule
  init_history(history);  //initialize history
  j_value=compute_jitter(instance_list);
  best_jvalue=jvalue;  // used by aspiration criterion
  nmoves=0;   //moves made so far
  for(i=1; i<=maxtry;i++) {        // maxtry > maxmoves
        taskinstance=select_taskinstance(instance_list); /* select a task */
        neighborhoodset=find_neighborhood(instance_list,taskinstance, neigh_size);
        evaluate_neighborhood(neighborhoodset,instance_list);
        best_move_record=select_best_valid_move(neighborhoodset, history, bestjvalue);
        if(best_move_record.exist)  {
            apply_move(best_move_record,instance_list);
            nmoves++;
            if (best_move_record.jvalue < bestjvalue)
               bestjbalue= best_move_record.jvalue;
           update_history(history,best_move_record);
        }
      if nmoves>maxmoves break;
  }
}
```

Fig. 2: The Core of Tabu Search Task Scheduling Algorithm

takes control of one of the CPUs. The jitter J_{ik} for the kth instance of task t_i is defined as:

$$J_{ik} = |s_{ik+1} - s_{ik} - P_i| \text{ for } k = 1,2,\cdots m_i - 1 \qquad (1)$$

$$J_{im_i} = |s_{i1} + lcm(P) - P_i - s_{im_i}|$$

The distributed scheduling problem can now be formulated as a combinatorial optimization problem on the set of all task instances. Our objective is to minimize the maximum jitter of all the tasks subject to precedence, exclusion, and deadline constraints and that no task can be scheduled before its release time, i.e.

Minimize $C = \sum_{t_i} \max_k \{J_{ik}\}$

Where J_{ik} is the jitter of kth instance of task t_i computed according to equation (1).

Tabu Search: In previous section , we formulated our scheduling problem as a combinatorial optimization problem. Tabu search is a well-known iterative procedure for solving discrete combinatorial optimization problems. It was first suggested by Glover (1990) and since then, it has been successfully applied to obtain optimal and suboptimal solutions to such problems as scheduling, time tabling, travelling salesman, and layout optimization.
A general framework of Tabu search is given in Fig. 1. Tabu search starts from some initial solution and attempts to determine a better solution in a manner of a "greatest descent neighborhood" search algorithm. The basic idea of the method is to explore the search space of all feasible solutions by a sequence of moves. A *move* is an atomic change, which transforms the current solution into one of its neighboring solutions. Associated with each move is a *move value*, which represents the change in the objective function value as a result of the move. Move values generally provide a fundamental basis for evaluating the quality of a move. At each iteration of algorithm, an admissible best move is applied to the current solution to obtain a new solution to be used in the next iteration. A move is applied even if it is a non-improving one, i.e. it does not lead to a solution better than the current solution. To escape from local optima and to prevent cycling, a subset of moves is classified as tabu (or forbidden) for certain number of iterations. The classification depends on the history of the search, particularly as manifested in the *recency* or *frequency* that certain moves of solution components, called *attributes*, have participated in generating past solutions. A simple implementation, for example, might classify a move as tabu if the reverse move has been made recently. These restrictions are based on the maintenance of a short-term memory function (the *tabu list*), which determines how long a tabu restriction will be enforced, or alternatively, which moves are admissible at each iteration. The *tabu tenure* (i.e. the duration for which a move will be kept tabu) is an important feature of tabu search, because it determines how restrictive is the neighborhood search.
The tabu restrictions are not inviolable under all circumstances and a tabu move can be overridden under some conditions. A condition that allows such an override is called an *aspiration criterion*. For example, an aspiration criterion might allow overriding a tabu move if the move leads to a solution, which is the best obtained so far.

Task Scheduling with Tabu Search: The basic tabu search is briefly discussed in the previous section. In this section, we specialize the basic tabu search into a specific algorithm for the task scheduling problem. This implies turning the abstract concepts of tabu search, such as initial solution, solution space, neighbourhood, move generaration, tabu criteria and others, into more concrete and implementable definitions.
The pseudocode for the proposed tabu search algorithm for task scheduling is given in Fig. 2. The input parameters to the algorithm are the descriptors of all the task instances in the scheduling time (input parameter instance_list), the maximum number of successful moves the algorithm makes before terminating (input parameter *maxmoves*), and the neighborhood size to be searched in each iteration (input parameter neigh_size). The former are ordered in a list, each descriptor containing information about the timing data, the precedence relations, the processor allocation, and the resource usage of the corresponding task instance.

The routine **tabu_search** makes use of other routines to perform the changes in the solution space. These are **select_taskinstance()**, **select_change()**, **find_neighborhood()**, **evaluate_neighborhood()**, and **select_best_valid_move()**, and apply_move. The compute_jitter routine calculates the jitter value of a schedule. The routine **init_scedhule()** finds the initial schedule to be improved by the algorithm. The routine **init_history()** initializes the history and the **update_history()** updates the history after a successful move.
The final output of the algorithm is an execution time (expressed as stating and a corresponding ending time) for each task instance in the scheduling period. These execution time intervals can be stored in a table to be used by run time dispatchers.
It is important to correctly define the parameters of the algorithm and to implement it as efficiently as possible to reduce the overall complexity and computation time. In the following sections, we describe how the important routines can be implemented.

Initial Solution: The routine **init_schedule()** uses the method proposed in (Natal and Stankovic, 2000) to obtain the initial solution required by the tabu search algorithm. The method obtains the initial solution by assigning the release times to the tasks in the following way:

$$\forall t_i : r_i < r_j \text{ whenever } t_i \rightarrow t_j$$

that is,

$$\forall t_i : r_i = \max\{r_i, \max_j\{r_j + c_i \text{ for all } t_j \to t_i\}\}$$

This method makes sure that the initial solution satisfies the precedence constraints, which is then maintained by the move operator (discussed below). Note that this method is not sufficient to satisfy the precedence constraints, as it does not guarantee, by itself, that the tasks will be scheduled in the right order. We must remember that shared resources and remote precedence constraints characterize our schedule. The move operator then applies the moves, which not only satisfy the precedence constraints but also other constraints.

Move Generation: For our scheduling algorithm, the set of all possible solutions consists of all the possible permutations of the tasks subject to the constraints. To generate a new schedule, it is possible to change the scheduling policy, or to keep the scheduling policy fixed and change the task parameters to new values consistent with their constraints. As we mentioned in an earlier section that each task has a task configuration consisting of its temporal characteristics (i.e. release times, deadlines, and computing time) the resources requested by each task and the precedence constraints. Therefore, a move to a neighborhood solution can be made by changing this configuration. Since it is only the release time of a task instance that can be changed, a new task configuration can be obtained by changing the release times of the tasks.

The choice of the new release times is not arbitrary, but a change is selected if it is *compatible* with the original release time, the deadlines and the precedence constraints. By compatible release time $r_{i'}$, we mean a release time r_i of task t_i would not prevent the feasible scheduling of the task. That is

$$r_{i'} \geq r_i$$

$$r_{i'} \geq r_j + c_j \text{ for all } t_j : t_j \to t_i$$

$$r_{i'} \leq d_i - c_i$$

$$r_{i'} \leq d_k - c_k - c_i \text{ for all tasks } t_k : t_i \to t_k$$

The algorithm generates a move in four steps. First, **select_taskinstance()** routine randomly selects a task from the *instance_list*, then **find_neighborhood()** finds a subset of neighborhood by changing the release time of the selected task, then **evaluate_neighborhood()** finds the jitter value for each move in the neighborhood set. The routine **select_best_valid_move()** selects the best valid move which is applied to obtain the next solution in the solution search space. Of course, as the whole neighborhood may be too large to be examined in one iteration, the proposed algorithm examines a sub set, whose size is determined by *neigh_size* parameter, of the whole neighborhood. This makes the algorithm more efficient without compromising the solution quality as our simulation results confirm it.

The routine **find_neighbourhood()** works as follows:

- First, it computes a time interval within which the release time of the task instance (selected by **select_taskinstance()** routine) can be moved. The beginning release time of the compatibility interval is given by the maximum among the actual release time for the task instance t_i and maximum among the earliest possible completion times $(r_j + c_j)$ of the predecessors t_j. The ending release time of the interval is the minimum among the latest possible release time for the task (i.e. $d_i - c_i$) and the deadlines of the successors t_k minus the sum of the times $c_i + c_k$ necessary to complete the execution of t_i and to execute t_k before its deadline.
- It then generates neighborhood set by randomly choosing new release times to the given instance.

Selecting a move by the method described above ensures that a valid move will preserve the precedence constraint, if such a move exits. If a valid move exits, then it then applied to the current solution by calling routine **apply_move()**.

Tabu Lists: The chief mechanism for exploiting memory in tabu search is to classify a subset of moves in the neighborhood as forbidden (tabu). This classification depends on the history of the search, particularly manifested in the recency or frequency that certain moves or solution components have participated in generating past solutions. We use the following tabu criteria to make certain moves tabu.

Recency Tabu: A move in our algorithm is characterized by $(t_i, r_i \to r_{i'})$, where $r_{i'}$ is the modified release time. A move $(t_i, r_i \to r_{i'})$ is prohibited (or is tabu) if its reverse move $(t_i, r_{i'} \to r_i)$ has been executed recently. Unlike standard tabu search in which the value of tabu tenure is fixed, we determine the tabu tenure of a move through a feedback (reactive) mechanism during the search. The tabu tenure of a move is equal to one at the beginning (the inverse move is prohibited only at the next iteration), and it increases only when there is evidence that diversification is needed, and it decreases when this evidence disappears. In detail, the evidence that *diversification is needed is signaled* by the repetition of previously visited configurations. All configurations found during the last *I* iterations of the search are stored in memory (use of a queue is a possible implementation strategy). After a move is executed, the algorithm checks whether the current configuration has already been found and it reacts accordingly (tabu tenure of the move increases if a configuration is repeated, it decreases if no repetition occurred during a sufficiently long period).

Same Jitter Value: This is a powerful tabu to diversify the search when stuck in a local optima. This tabu is triggered if the same jitter value has been obtained during the last c iterations, where *c* is specified by the user. When the tabu is switched on, all solutions with a cost selected during the period the tabu is active are tabued. The user can also specify the tenure of the tabu.

Aspiration Criterion: Tabu conditions based on the activation of some moves attributes may be too restrictive and result in forbidding a whole set of unvisited moves which might be attractive. The aspiration criterion allows overriding a tabu move under certain conditions. The proposed algorithm overrides a tabu restriction if the move leads to a solution better than the best found so far. This is the only aspiration criterion used by the proposed algorithm.

Results and Discussion

Since no benchmarks exist for the real-time scheduling problem discussed in this paper, we resort to synthetic test cases that span a large parameter space. We mainly investigated the performance of the algorithm for various utilizations, varied harmonic relationships of the periodic tasks, and varied period to deadline ratio.

Test data used in our simulation were similar to that used in (Natal and Stankovic, 2000). Briefly, maximum number of processes was taken 20 with 1 to 3 tasks in each process with a maximum of 200 tasks. Minimum number of resources required by a task was set to 2 and maximum to 3. The probability that a task will use a resource was set to 0.4. We performed our simulation on three classes of processes (and tasks) according to their periods.

In class 1, periods were randomly drawn from 16, 32, 64, 128, 256 (i.e. harmonic instances). The experiments in this class were further divided into three groups: the first was with utilization $0.2 \le U \le 0.5$ (deadline to period ratio = 1 to 0.2), the second with $0.3 \le U \le 0.75$, the third with $0.55 \le U \le 0.85$. In class 2, the periods were selected from the set 10, 20, 50, 100, and 180. The experiments in class 2 were divided into two groups: the first with $0.3 \le U \le 0.75$ (deadline to period ratios from 0.25 to 1), the second with $0.55 \le U \le 0.9$. The periods for class3 were drawn from the set 15, 30, 50, 180 and 200. The first group in this had $0.25 \le U \le 0.5$ (deadline to period ratios from 0.25 to 1) and the second group had $0.3 \le U \le 0.65$. After conducting a series of preliminary experiments, neighborhood of size 5 was found to produce best results. Since an increase in the neighborhood size did not improve the solution quality (on average), all the results discussed below are for neighborhood of size 5. Number of maximum moves (parameter maxmoves) was set between a value from 1000 to 20000 depending on the problem size.

A total of 45 runs on task sets belonging to class 1 were performed. The results are shown in table 1. The first column shows various values of utilization *U*, the second column shows the number of task sets successfully scheduled/total number of task sets. The percentage of successfully scheduled task sets is also shown in parentheses. Average jitter value (in time units) for successfully scheduled task sets is shown in column 3. The results show that the algorithm successfully scheduled 95.55% of task sets with an average jitter value 2.48.

Table 1: Results for Class 1

U	Scheduled set/total set	Average jitter
$0.2 \le U \le 0.5$	15/15 (100%)	0.62
$0.3 \le U \le 0.75$	15/15 (100%)	1.98
$0.55 \le U \le 0.85$	13/15 (87.66%)	4.83
Overall	43/45 (95.55%)	2.48

A total of 30 run were performed for class 2 tasks. The results are shown in table 2. The proposed algorithm was able to successfully schedule 29 task sets out of 30 with an average jitter value of 6.53. Results for class 3 task sets are shown in table 3. The proposed algorithm found feasible schedules of 15 task sets out of 20 task sets. The average jitter value was 7.42.

Table 2: Results for Class 2

U	Scheduled set/total set	Average jitter
$0.3 \le U \le 0.75$	15/15 (100%)	1.27
$0.55 \le U \le 0.9$	14/15 (93.33%)	5.26
Overall	29/30 (96.67%)	6.53

Table 3: Results for Class 3

U	Scheduled set/total set	Average jitter
$0.25 \le U \le 0.5$	8/10 (80%)	5.20
$0.3 \le U \le 0.65$	7/10 (70%)	9.63
Cverall	15/20 (75%)	7.42

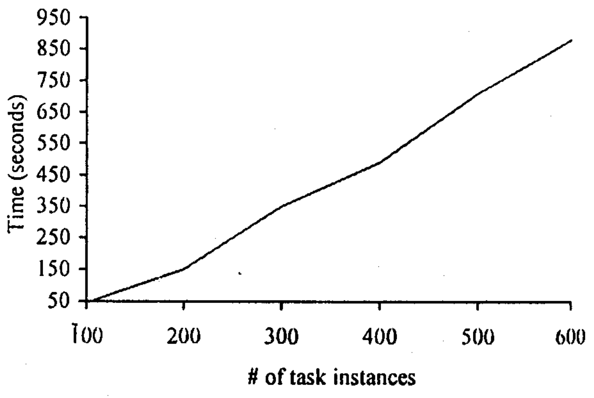

Fig. 3: Computation Time for Tabu Search Scheduling Algorithm

The run time cost of executing the algorithm is strongly dependent on the complexity and size of the task set. The average run time for various sizes of task sets is shown in Fig. 3 when the program written in C was executed on Pentium IV processor. Even though the execution times are quite high but for static scheduling systems, the schedules are computed once only and stored in a table latter to be used by the local dispatchers. A reasonable approach to using tabu search algorithm is to limit the use of the algorithm for scheduling only the time critical and jitter sensitive tasks and then use the time gaps left by those tasks to execute the non real-time tasks.

Conclusion

Tabu search algorithm has been used successfully to solve many combinatorial optimization problems. In this paper, we presented a tabu search approach for scheduling real-time tasks with minimum jitter in a distributed computing environment. We have presented methods to obtain initial solution, move generation, tabu and aspiration criteria. We have shown the performance of the tabu search scheduling algorithm through a simulation study. Experimental results demonstrate the effectiveness of the proposed algorithm to the real-time task scheduling problem.

References

Bate, I. A.Burns, *et al.*, 1996. Towards a Fixed Priority Scheduler for an Aircraft Application. Proc. Euromicro Workshop on Real-Time Systems.

Burns, A. *et al.*, 1993. The Olympus Attitude and Orbital Control System: a Case Study in Hard Real-time system design and implementation, Technical Report YCS90, Dept. of Computer Science, Uni. Of York.

Carpenter, T. K. Driscoll, *et al.*, 1994. ARINC 659 Scheduling: problem definition, Proc. 1994 Real-Time Systems Symp.

Fohler, G., 1994. Flexibility in Statistically Scheduled Hard Real-time Systems, Technische Naturwissenschaftliche, Fakultaet. Technische Uni. Wein.

Glover, F., 1990. Tabu Search: a Tutorial. Interfaces. 20: 74-94.

Hou, E. S. N. Ansari and H. Ren, 1994. Genetic Algorithm for Multiprocessing Scheduling. IEEE Trans. Parallel and Distributed Systems. 5: 113-120.

Hubscher, R. and F. Glover, 1996. Applying Tabu Search with Influential Diversification to Multiprocessor Scheduling. Computers and Operations Research. 21: 877-884.

Kidwell, M. D. and D. J. Cook, 1994. Genetic Algorithm for Dynamic Task Scheduling. IEEE 13th Annual Int. Phoenix Conf. on Computers and Communications. 61-67.

✓ Manimaram, G. and C. S. Murthy, 1998. An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems. IEEE Trans. Parallel and Distributed Systems. 9: 312-319.

Mahmood, A., 2002. A Tabu Search Algorithm for Scheduling Real-time Tasks under Precedence and Resource constraints. Int. J. Science Vision (to appear).

✓ Mahmood, A., 2000. A Hybrid Genetic Algorithm for Task Scheduling in Multiprocessor Real-time Systems. Int. J. Studies in Informatics and Control. 9: 207-217.

✓ Natale, M. D. and J. A. Stankovic, 2000. Scheduling Distributed Real-time Tasks with Minimum Jitter. IEEE Trans. Computers. 49: 303-316.

Porto, S. C. S. and C. C. Ribeiro, 1995. A Tabu Search Approach to Task Scheduling on Hetrogeneous Processes under Precedence Constraints. Int. J. of High Speed Computing. 17: 21-40.

Ramamritham, K., J. A. Stankovic and P. F. Shiah, 1990. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. IEEE Trans. Parallel and Distributed Systems. 1: 184-194.

Stankovic J. A., K. Ramamritham and S. Cheng, 1985. Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-time Systems. IEEE Trans. Computers. 34:1130-1143.

Tindell, K., K. A. Burns, and A. Wellings, 1992. Allocating Real-time tasks (an NP-hard Problem made easy). Real-Time Systems J.

Xu, J. and D. Parnas, 1990. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. IEEE Trans. Software Engineering. 16: 360-369.

Y. C. Kim and Y. S. Hong, 1993. Task Allocation using Genetic Algorithm in Multiprocessor Systems. Proc. 10th IEEE Regional Conf. on Computers. Communications, Control and Power Eng. NJ. 258-261.