Providing for Dynamic Arrivals during the Static Allocation and Scheduling of Periodic Tasks

K. Ramamritham and J. M. Adan

COINS Technical Report 90-107

October 24, 1990

Providing for Dynamic Arrivals during the Static Allocation and Scheduling of Complex Periodic Tasks[1]

Krithi Ramamritham
University of Massachusetts
Amherst, MA 01003
USA

Juan Manuel Adan
Centro Tecnologico para Informatica
Cx. Postal 6162 - CEP 13081
Campinas, SP - Brazil

October 24, 1990

Abstract:

Resources needed to meet the deadlines of safety-critical tasks are typically preallocated. Also, these tasks are usually statically scheduled such that their deadlines will be met even under worst-case conditions. Besides *periodicity constraints*, these algorithms should be able to handle *resource*, *precedence*, *communication*, and *fault tolerance constraints.* Most real-time systems consist of such statically scheduled tasks as well as tasks that arrive dynamically. Thus, provision should be made during static scheduling to cater to the needs of dynamic arrivals. Static schedules aimed at just the feasibility of periodic tasks have two characteristics that affect the schedulability of dynamic arrivals: (1) There is a lack of load balance across the processing elements; (2) Idle times appear in a clustered manner. In this paper, we investigate two approaches to address these problems. One involves specifying a limit to the load imposed on a node during the allocation of tasks to processors and nodes. The other forces an idle time interval following the completion of a task. Effect of different limits on the loads as well as task execution time dependent idle times are studied via simulation. The experiments indicate that these very simple techniques can be used to enhance the load balance characteristics without jeopardizing the schedulability of the static periodic tasks.

[1]This work was supported by ONR under contract NOOO14-85-K-0389, by NSF under grant DCR-8500332, and by a grant from Texas Instruments.

1 Introduction

In providing scheduling support for real-time systems, it is important to take into consideration the nature and characteristics of tasks in a given system. In a system interacting with an environment that is dynamic, large, complex, and evolving, many types of tasks exist [18]. These can be categorized based on their interaction with and impact on the environment. *Safety-critical* tasks are those real-time tasks which must meet their deadline, otherwise a catastrophic result might occur. It must be shown *a priori* that these tasks will always meet their deadlines subject to some specified number of failures. Thus, resources must be allocated and scheduling decisions must be made such that these tasks will always meet their deadlines. These decisions are made under worst-case assumptions of arrival rate, execution times, and resource needs. In other words, the safety-critical tasks are treated as periodic tasks.

We have already developed an algorithm that is suitable for the static allocation and scheduling of complex, safety-critical periodic tasks. Besides periodicity constraints, tasks handled by the algorithm can have resource requirements and can possess precedence and communication, as well as fault tolerance constraints [15]. The primary criterion in the static scheduling of periodic tasks is *feasibility*, i.e., determining a feasible schedule wherein all tasks meet their timing requirements, precedence constraints, etc. Because the allocation and scheduling problem being considered is NP-hard in the *strong* sense [5], our algorithm is heuristic in nature.

For a vast majority of tasks in a real-time system, a catastrophe will not result if they are not finished on time. But system performance will degrade if their timing constraints are not met. It is necessary to treat such *essential* tasks in a dynamic manner as it is impossible to reserve enough resources for all contingencies with respect to these *nonperiodic* tasks.

In some systems, in order to isolate the safety-critical components, a separate subsystem could be earmarked for safety-critical tasks. An independent subsystem will have to be provided for essential tasks. Where this is not possible, that is, if both periodic tasks as well as nonperiodic tasks exist in a system, provision should be made during static scheduling to cater to the needs of dynamic arrivals. That is, the allocation and schedules produced for safety-critical tasks should be carried out such that the schedulability of essential tasks is improved. This is the motivation for the work described in this paper.

Suppose we consider a distributed system of multiprocessor sites. When static schedules for distributed or multiprocessor systems are examined, two types of phenomena can be observed. The first relates to the computational loads imposed on the individual sites or processors. Typically, there exists a load imbalance within the processors on a site as well as across the sites. The second relates to the idle times on individual sites or processors. Often, these occur in a clustered fashion towards the end of the schedule, that is, idle times are not evenly spread across the schedule. Both of these are to be avoided if the idle time and resources are to be used to better accommodate dynamic arrivals. In particular, we would like to balance the loads not only across individual sites and processors within a site but also along the time axis.

We have investigated two approaches to address these problems. One involves defining a limit to the load imposed on a site during the allocation of tasks to processors and sites. The other forces an idle time interval following the scheduled execution of a task. Effect of different limits on the loads as well as task execution time dependent idle times have been studied. The experiments indicate that these very simple techniques can be used to enhance the load balance characteristics without jeopardizing the schedulability of the static periodic tasks. In addition, more flexibility can be gained by modifying, where possible, the start times of the components of critical tasks allocated to a site. If this modification is done such that the ancestors of these components on other sites are not delayed or that task deadlines are missed, more dynamic arrivals can be guaranteed.

A majority of current allocation and scheduling work dealing with periodic tasks assume that tasks have only periodicity constraints [9], [4], [1], [16]. Constraints such as those that arise from resource sharing and synchronization are beginning to be considered [17], [11]. Resource constrained scheduling is also investigated in [19], [20], but they deal with periodic tasks that execute on a single site in a distributed system. [6] deals with the allocation and scheduling of simple periodic tasks having fault tolerance requirements on a multiprocessor.

Whereas a lot of work has been done in the area of load balancing in distributed systems (see [2] for a survey), there is little extant work on load balancing in distributed real-time systems. The algorithm reported in [1] aims to balance the loads across the sites while allocating replicated periodic tasks in a distributed system. Here periodic tasks are independent, simple entities, without precedence or other constraints. The only requirement is that replicates of a task be on different sites. In [7], techniques are proposed for improving the *responsiveness* to dynamic task arrivals in the presence of periodic tasks scheduled according to rate-monotonic priority assignment [16]. Here again, tasks are simple and independent and the focus is a single processor system. In contrast to these related work, we are concerned with load balancing issues when complex periodic tasks have to be allocated and scheduled across a distributed system.

This paper is organized as follows: the system and task models are defined in Section 2. Section 3 provides an overview of the static allocation and scheduling algorithm and gives an example of the two problems mentioned earlier. Strategies for addressing these problems are detailed in Section 4. These strategies are evaluated in Section 5. Gaining further flexibility for scheduling dynamic arrivals, through dynamic adjustments to the static schedules is the subject of Section 6. Concluding remarks are made in Section 7.

2 Task and System Characteristics

We consider distributed systems consisting of a number of sites, with a set of resources attached to each site. We assume communication media and protocols that have predictable communication delays such that knowing the arrival time and characteristics of a message, we can predict when the message will be delivered [13]. For instance, point-to-

point networks or multi-access networks employing protocols such as 803.2D [8] have such predictability.

To simplify the following discussions, the assumption is made that the sites are connected by a *multiple-access network*. Communication from one site to another occurs at prespecified times, as per the schedule generated. To be more precise, a set of consecutive time slots of the communication channel is dedicated exclusively for communication between a particular pair of subtasks. (This is similar to allocating a set of consecutive time slots of a processing element exclusively for executing a subtask.) Since the scheduler preschedules the communication on the network, no contention occurs for the multiple access network at run time. A reader may desire to examine Fig 2 to get some idea about how such a schedule looks. (In Section 6 we examine techniques for dynamically rescheduling the communication without jeopardizing the schedulability of the periodic tasks.)

The different characteristics of tasks and the rationale for their consideration are explained in [15]. They include:

1. The period of the task. One instance of all subtasks of a task must be executed every period.

2. Subtasks are nonpreemptable.

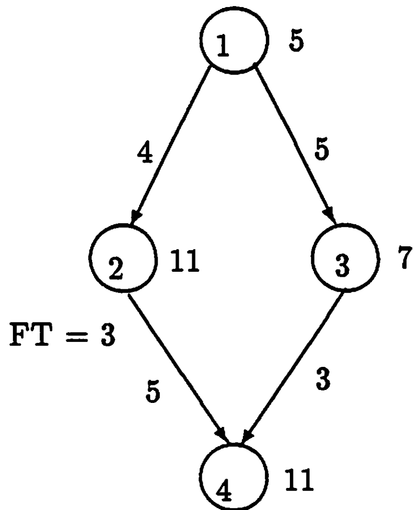
3. The precedence relationship among subtasks is expressed as a graph wherein the nodes represent subtasks and a directed arc exists from a subtask to its successor.

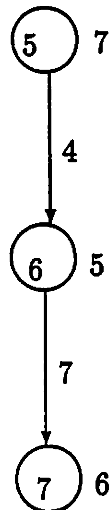
4. Computation times of subtasks are expressed via values attached to each graph node. These represent worst-case computation times.

5. The maximum amount of information communicated from a subtask to its successor is expressed via a value attached to the corresponding arc. For simplicity, we assume that the value associated with arcs in the graph are the communication times for the corresponding messages.

6. The fault tolerance requirements of subtasks is specified by a value attached to each subtask indicating the number of replicates needed for the subtask.

In addition, resource constraints could be attached to each subtask express any specific resources needed by that subtask. These include the CPU, sensors, I/O devices, data structures, files, and data bases. It is assumed that all the specified resources are needed by the subtask throughout its execution and that resources allocated to a subtask are released at the end of its execution. The latter implies that resources allocated to a subtask are not held over for its successor. (With some minor changes, the algorithm can be made to handle the situation where a subtask releases some of its resources early.) The resource constraints restrict the sites to which a subtask can be assigned: these sites should have the resources required by the subtask. In this paper we assume that a site has just one processor and confine our attention to just two resources: the processor on a site and the communication network.





Periodic Task #1:
Subtasks = 1, 2, 3, and 4
Period = 100
Subtask 2 is required to be executed in triplicate; Subtask 4 votes on the results communicated by the three copies of 2.

Periodic Task #2:
Subtasks = 5, 6, and 7
Period = 50

The number to the right of a node indicates the computation time of the subtask corresponding to the node. The number attached to the arc connecting two subtasks indicates the amount of information communicated from one subtask to its successors.

Figure 1: Structure of two Periodic Tasks

Turning our attention to dynamically arriving tasks, we assume that these tasks have characteristics similar to that of periodic tasks but for one exception: instead of the periodicity constraint, they have a deadline constraint. When a set of precedence-related subtasks have a common deadline, all the subtasks should be completed by that deadline. Such tasks may arrive at any site in the system. The dynamic scheduler on a site will handle the scheduling of that task on that site and across the distributed system using distributed scheduling algorithms that take task timing constraints into account [12], [3]. Studies of these algorithms show that even though a system need not be perfectly balanced to achieve the best performance, the better the load balance, the higher the number of tasks that can be scheduled to meet their deadlines.

In this paper, we focus on how such load balance can be achieved given that we have a base set of statically allocated and scheduled tasks. Thus, our goal is to come up with strategies, that can be incorporated into a static algorithm, for balancing the load among the processing elements.

3 Overview of the Static Scheduling and Allocation Algorithm

Given a set of periodic tasks, the algorithm attempts to assign subtasks of the tasks to sites in a distributed system and to construct a schedule of length L where L is the least common multiple of the task periods. A real-time system with the given set of tasks then repeatedly executes its tasks according to this schedule every L units of time.

The algorithm consists of four steps. Step-I constructs the *comprehensive graph* containing all instances of the tasks that will execute in an interval of length L. In Step-II, the comprehensive graph is embellished with replicates of the subtasks that have fault-tolerance requirements. This results in the addition of some subtasks and some arcs to the graph. Step-III involves clustering subtasks in the comprehensive graph. Specifically, based on the amount of communication involved between a pair of communicating subtasks and the computation time of the subtasks, a decision is made as to whether the two subtasks should be assigned to the same site, thereby eliminating the communication costs involved. The algorithm makes its decision based on whether the fraction $\frac{\textit{sum of the computation time of the two subtasks}}{\textit{cost of communication}}$ is lower than a tunable parameter called the *communication factor*, CF. Applying the above scheme to *every pair* of communicating subtasks in the comprehensive graph derived in Step-II, a *communication graph* is generated with the current value of CF. Step-IV allocates the subtasks to sites in the system, allocates the communication (between subtasks) to the time slots in the communication channel, and if possible, determines a feasible schedule. This is done using a heuristic search technique that takes into account the various task characteristics, in particular, subtask computation times, communication costs, deadlines, and precedence constraints. It allocates a subtask to a site, determines the order in which each site processes its subtasks, and schedules communication. The allocation and scheduling decisions are made in a coordinated fash-

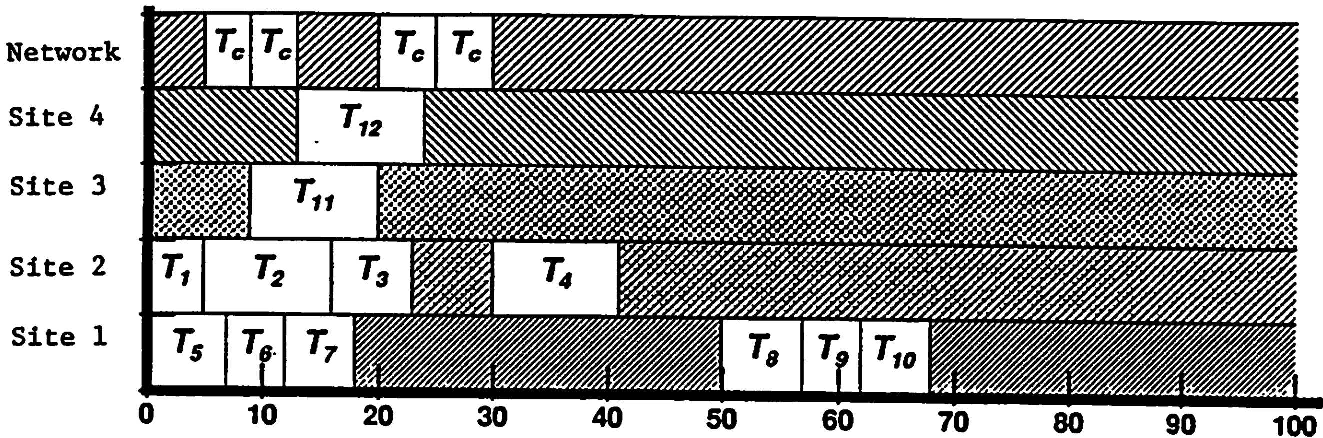


Figure 2: One Possible Schedule

ion. Specifically, allocation and scheduling decisions about a subtask are made only after all its predecessors have been allocated and scheduled. Of course, these decisions take into account the communication and computational needs of the subtasks that follow. If, at the end of Step-IV, a feasible allocation and schedule is not possible, the value of CF is altered, and Steps III and IV are repeated. In [15], we provide details of each of these steps.

Figures 2 and 3 show two different schedules produced by this algorithm when applied to the tasks in Figure 1. Here subtasks 11 and 12 are the replicates of subtask 2 which is expected to be triplicated. Subtasks 5, 6, and 7 (8, 9, and 10) are subtasks of the first (second) instance of the periodic task #2. All intertask communication, indicated by T_c, are scheduled on the multiple-access channel. The schedule in figure 2 was generated with CF = CFmax, a CF value that assigns all communicating subtasks (except replicates) to the same site. The schedule in figure 3 was generated with CF = 0. This assigns communicating subtasks to different sites. (Here we did not constrain two instances of a particular subtask (subtasks 5 and 8, for example) to be allocated to the same site. However, if the last subtask and the first subtask of a periodic task are not allocated to the same site, a communication is scheduled on the channel to communicate the results of the one instance of a periodic task to the next.)

We chose to compare the performance of different algorithms or different parameter settings using the *Success Ratio* metric. If an algorithm is able to find feasible schedules for X of the given Y task sets, (where none of the Y task sets are definitely infeasible) its success ratio is said to be (X/Y). We express this typically as a percentage.

Given the computational intractability of the allocation and scheduling problem at hand, it is not practical to compare a given heuristic approach with an optimal algorithm. Hence the Success Ratio is used for comparing competing (heuristic) strategies. Obviously,

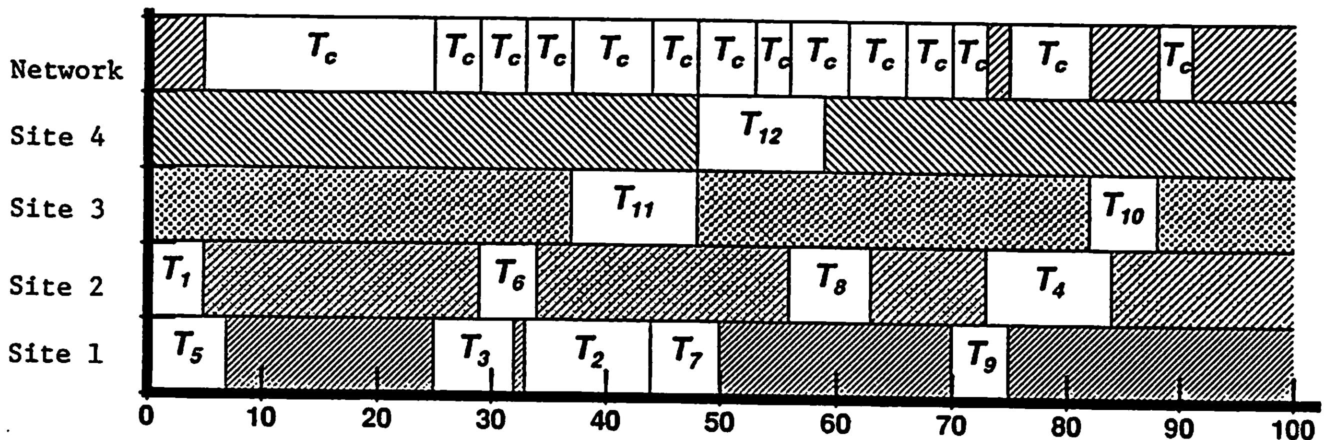


Figure 3: Another Schedule

a strategy that has a higher success ratio than another is the preferable one.

We also measured the overhead of an algorithm in terms of the average value of the *total* number of points (in the search trees) constructed by the algorithm in the process of determining the allocation and schedules for subtasks in a given task set. This – rather than the actual time – is a reasonable choice since it shows the abstract cost characteristics of the algorithm. Also, for the cases where comparison is made, the cost of decision making at each point in the search is the same.

Our experiments indicate that the algorithm constructs the search path so effectively that if the initial path does not lead to a feasible schedule for a given set of tasks, chances are high that we have an infeasible task set. This is attested by our test results which show that even a large number of additional backtracks produces only a small marginal improvement in performance.

4 Strategies for Improving the Load Balance

Two heuristic strategies have been developed to produce a well balanced schedule. The first is called MLS, the Maximum Load Strategy, and the other is called GFS, the Gap Factor Strategy.

In MLS, allocation decisions made by the algorithm are constrained by the maximum load that is permitted to be imposed on each site. The algorithm tries to produce an allocation where each site has no more than *MaxLoad*% of the total load of the task set allocated to it. If it is not possible to schedule a given task set using the current value of *MaxLoad*, a new one is tried. This new value is obtained by adding *MaxLoadInc* to *MaxLoad*. This process is repeated until the task set is successfully scheduled or *MaxLoad*

```
Initialize MaxLoad; (* <= 100% *)
Initialize GapFactor; (* >= 0 *)
Initialize GapFactorInc, MaxLoadInc;
Success := false;
WHILE ((MaxLoad <= 100) AND NOT Success)
BEGIN
    CurrentGapFactor:=GapFactor;
    WHILE ( (CurrentGapFactor >= 0) AND NOT Scheduled)
    BEGIN
       Success = <Attempt a Feasible Allocation and Schedule>
       CurrentGapFactor := CurrentGapFactor - GapFactorInc;
    END
    MaxLoad:=MaxLoad + MaxLoadInc;
END
```

Figure 4: Pseudo code for the Overall Algorithm

reaches 100%. Thus, the initialization of *MaxLoad* with 100% is equivalent to running the algorithm with this strategy disabled.

In GFS, the scheduling algorithm introduces *gaps*, i.e., idle times, after each scheduled task. When gaps are introduced, the number of subtasks that can be allocated to a site decreases, forcing the algorithm to try to distribute the tasks over the system. This should help in improving the load balance. *GapFactor* indicates the percentage of the computation time that should be left idle after each scheduled task. Each time the algorithm fails to find a schedule with a given *GapFactor*, it is decreased by *GapFactorInc*%. This process is repeated until the task set is successfully scheduled or *GapFactor* reaches zero. Setting the initial value of *GapFactor* to zero disables this strategy.

Figure 4 presents the pseudo-code for the algorithm. The two factors, *MaxLoad* and *GapFactor* are used to capture the needs of MLS and GFS respectively.

MLS is applied when the initial value of *MaxLoad* is set to a value less than 100%. In the same way, GFS is applied when *GapFactor* is initialized to a value greater than zero. Thus, the algorithm applies one of the strategies or both, depending on the initial values of *MaxLoad* and *GapFactor*.

Given these strategies, the static scheduling algorithm developed in [15] has to be modified in the following ways:

- Suppose a subtask is scheduled to start on site p at time s and has a computation time of c. If the current *GapFactor* is g, then another subtask can be scheduled to start on p only after $(s + g \times (1 + c))$.

- Subtasks can be allocated on a site as long as the load on that site does not exceed

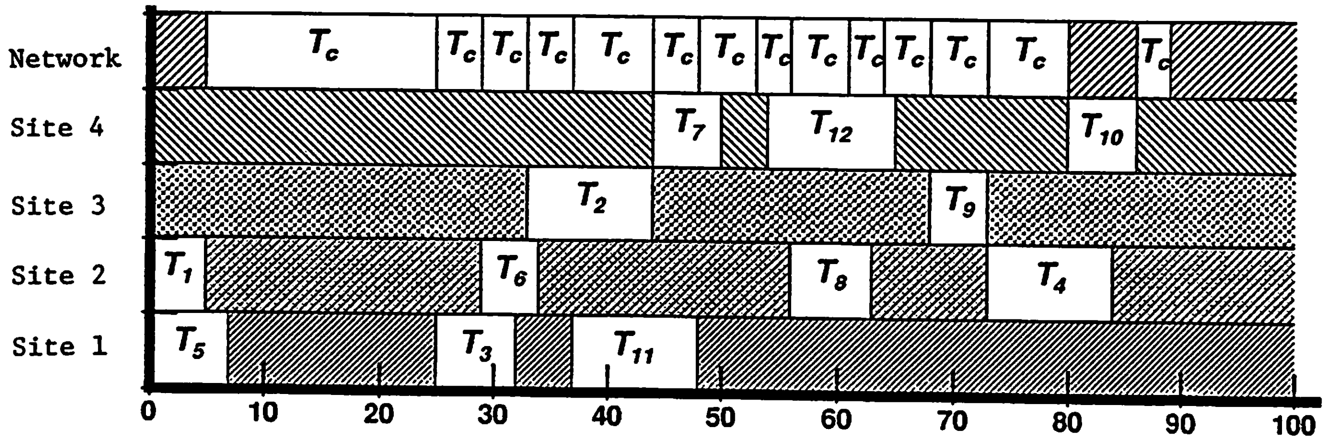


Figure 5: Balanced Schedule

MaxLoad.

To determine the effect of the load balancing strategies, we use a metric called *load deviation* defined as:

$$\sum_{i=1}^{\#sites} (|(AvgLoad - Load_i)|)$$

where *AvgLoad* is the average load across the sites and $Load_i$ is the load allocated to $site_i$. Both are expressed as a percentage of the total load imposed by the task set being considered for scheduling. The load deviation is an indication of the lack of load balance in a given system. As can easily be observed from the expression above, the better the load balance, the lower is the load deviation.

Consider, for the sake of an example, that in a system with two sites 40% of the load corresponding to a given task set is allocated to the first site and the remaining 60% of the load is allocated to the second site. In this case, AvgLoad is 50% and the load deviation is given by abs(50-40)+ abs(50-60), that is 20%. If 50% of the computation time were allocated to each site, the system would have a perfect load balance and its load deviation would be 0.

Figure 5 shows the schedule produced when the MLS and GFS strategies are applied to the schedule in 3. The GapFactor for this schedule is 80%. The load deviation is 14. In contrast, the load deviation for the schedule in figure 3 is 34 and for the schedule in figure 2 is 48.

5 Experimental Evaluation of the Load Balance Strategies

In this section, we first discuss the experimental setup and then present results of various experiments.

Each point in the plots presented in this section was produced by running the algorithm on 100 different periodic task sets. The precedence structure of each periodic task was generated using the random graph generation package discussed in [10]. This package can be used to generate general graphs, trees, and chains. All our periodic tasks have precedence constraints that can be represented as general graphs and have the following characteristics:

- The computation time of each subtask is uniformly distributed between 50 and 100 time units.

- The communication cost attached to an arc in the precedence graph is $(comm_ratio \times C)$ where C is the average computation time of a subtask, i.e., 75. Experiments were conducted for *comm_ratio* values between 0.1 and 0.4.

- Recall that some of the subtasks in a task are replicated for fault tolerance. In the experiments, with a probability of *redundancy_ratio*, each subtask of a task has redundancy_no of redundant subtasks (i.e., a total of 1 + *redundancy_no* copies of a subtask). In the experiments, *redundancy_ratio* = 0.1 and *redundancy_no* = 1.

- To exercise the algorithm under different periodicity constraints, the following scheme was devised. A parameter, *laxity_factor*, was used to set the period, P, of the first task as follows:

  $$P = C \times task_size \times (1 + (redundancy_ratio \times redundancy_no)) \times laxity_factor \quad (1)$$

  Note that this formula is based on just the computational requirements of the tasks. The computational requirements of a task increase with the number of subtasks in a task, given by *task_size*, with the average computation time of a subtask, with the *redundancy_ratio*, and with the *redundancy_no*. Under the above formula, given a task with certain computational requirements, the larger the laxity_factor, the larger the period of the task and hence, larger the leeway for scheduling.

- Even though we have conducted experiments with larger task sets, all the results shown here are for task sets with three periodic tasks. The first periodic task has four subtasks (i.e., its task_size=4), the second has eight, and the third has twelve. Given the above formula for P, the period of the second task is twice that of the first; the period of the third is three times that of the first. Thus the length L of the schedule generated = the least common multiple of the three periods = $(6 \times P)$.

These periodic tasks produced comprehensive graphs with around 75 sites and depending on the communication factor (CF), *comm_ratio*, *redundancy_ratio*, and the *redundancy_no*, the communication graphs had between 100 and 150 sites.

Even though we generate 100 task sets to obtain each point in the plots, we removed from consideration task sets that were definitely infeasible. This was detected by determining the latest start time of the first subtask of a periodic task, ignoring all communication. If the latest start time of this subtask is less than 0 the task set is not considered further. (In no case did we find more than 25 out of the 100 tasks generated to be definitely infeasible, thus leaving at least 75 tasks for exercising the algorithm.)

The plots presented in this section represent the behavior of the allocation and scheduling algorithm when it was not permitted to backtrack to a previous search point upon failure. This was done since our previous studies indicated that if the initial search path taken by the algorithm does not lead to a feasible schedule, it is highly likely that the task set being dealt with is unfeasible.

In these experiments, when using MLS, *MaxLoad* was initialized to (100/#sites)+1 and *MaxLoadInc* was set to 3%. The number of sites defined in the experiment to produce the results discussed here is 6. Thus, that for this experiment AvgLoad is 17%, *MaxLoadInc* is 3%, and MLS uses the following values for *MaxLoad*: 17,20,23,...,100. For GFS, *GapFactor* is initialized to 100% and *GapFactorInc* is set to 10%.

5.1 Performance of the Load Balancing Strategies

We present experimental results that show the effect of the strategies on the success ratio, on the load deviation, and the values of *MaxLoad* and *GapFactor* that produced the resulting load balance. The results for two different communication factors, namely, 0.1 and 0.4, are shown.

In the plots, the graph marked "MLS" ("GFS") shows the results when only MLS (only GFS) is applied. The graph marked "neither" shows the results when neither stragety is applied, i.e., this graph is for the original allocation and scheduling algorithm. The graph marked "both" shows the results when both strategies are applied.

Plots 1 to 4 present the results obtained for task sets with communication ratio equal to 0.1. From plot 1, we can see that the load balancing strategies have negligible impact on the schedulability of tasks.

Plot 2 shows that the two strategies improve the load balance considerably. As expected, the combination of the GapFactor and the MaxLoad strategies gives the best load balance. However, the combined performance is only slightly better than that offered by the use of the MaxLoad strategy alone. On the other hand, the improvement in the load balance using MLS, compared with the case when none of the strategies are used, is very significant. Note that when none of the strategies are applied, the load deviation increases with increasing laxities. This is because, with increasing laxities, a given site is able to accommodate more tasks. This increases the load imbalance. When both strategies are

used, the opposite is the case: the load deviation decreases with increasing laxities.

Plot 3 shows the average *MaxLoad* used to generate the successful schedules for the given task sets. Note that *MaxLoad* in the plot does not correspond to the actual load allocated to each site; it represents the limit employed by the algorithm during the allocation process, i.e., the values of *MaxLoad* for which the algorithm could find a successful schedule, when used to limit the load allocated to each site. The reason for the decreasing *MaxLoad* with increasing laxities is as follows: when laxity increases, the overall load imposed on the system decreases resulting in a decrease in the average load imposed on a site. With increasing laxities, more communication overheads can be be accommodated and hence subtasks of a task can be better distributed. This produces a better load balance and hence a smaller value for MaxLoad. Plot 3 also shows that the needed *MaxLoad* is slightly smaller when MLS is applied in conjunction with GFS.

Plot 4 shows the *GapFactor* used for the generation of the successful schedules corresponding to plot 1. The combination of the GapFactor and MaxLoad strategies produces a lower gap between tasks than GFS alone. The reason for this is that as mentioned earlier, with increasing laxities, MLS distributes the subtasks among the sites. Because of the concomitant increase in the communication overheads, a subtask that has a successor on another site has to finish earlier (and hence will have to start earlier) compared to the situation when its successor is on the same site. This implies that the gap following the predecessor of this subtask will have to be smaller.

Plots 5 to 8 present the results obtained for task sets with communication ratio 0.4. With the reduction of the success ratio of the algorithm to find a schedule, associated with the demand for more communication capacity, comes a decrease in the system load balance. This is related to the fact that the higher communication costs between tasks lead the algorithm to cluster more tasks on the same site so as to avoid some communication. The other observations that relate the performance of the various strategies, *vis a vis* the load deviation, the *MaxLoad* used, and the *GapFactor* used, can be seen to be true for this larger communication ratio.

5.2 Cost of the Load Balance Strategies

Besides the quality of the generated schedules in terms of load balance, another important consideration when choosing a strategy for load balance is its computational cost. Plot 9 presents the cost of the algorithm, in number of search points created, for the failure cases, i.e., for the cases where the algorithm could not find a feasible schedule. Plot 10 presents the cost for the successful cases, i.e., for the cases where the algorithm could find a feasible schedule. (Note the difference in scale of the Y axis in the two plots.) From the plots, it is clear that the improvement in load balance, using the two strategies, produces a considerable increase in cost. In particular, the performance improvement of the combined MLS and GFS, over MLS alone, comes about at a disproportionate increase in cost. On the other hand, the GFS alone has only a very modest performance improvement considering its cost, particularly for task sets with low laxities. This is partially due to

the fact that independent of the task set laxity, the algorithm always tries to generate a feasible schedule beginning with a 100% *GapFactor*, which is evidently too high a value for tasks with lower laxities. The cost for the failure cases is higher because a task set is considered unschedulable only if all combinations of CF, *MaxLoad*, and *GapFactor* values fail to produce a feasible schedule. A given combination will be more successful as laxity increases. So if a task set is schedulable, a successful combination will be found sooner when laxity increases. This accounts for the decrease in costs with increasing laxities for the success cases.

Given the above discussion, it should be evident that the initial values chosen for *GapFactor* and *MaxLoad*, as well as the order in which different combinations are considered, will play a large part in the overall costs. In the tests done thus far, for a given *MaxLoad* value, different *GapFactor* values were tried and for a given (*MaxLoad, GapFactor*) combination, different CF values were tried.

It will be fruitful to try different heuristics for searching the available three-dimensional space for the appropriate values. The laxity of the tasks being scheduled, the communication costs *vs.* the computation times, and the overall task load information can be useful in this search.

The next experiment conducted proves this point. After running the algorithm for a number of cases, the scheduling parameters *GapFactor* and *MaxLoad* can be adjusted to reduce the number of points to visit during the search for a feasible schedule. This was done for the *GapFactor* strategy. The same task sets used to generate the results shown above were submitted to the algorithm, initializing *GapFactor* to the average values obtained in the previous experiment. If we consider the tuple, (laxity factor, Initial *GapFactor*) the set of tuples used on this new experiment was (0.9,10);(1.0,20);(1.1,30);(1.2,40);(1.3,50);(1.4,70); (1.5,80) and (1.6,90).

Plots 11 and 12 show the results obtained with this *adaptive* initial setting of the scheduling parameter *GapFactor*. We can observe in the plots that this setting has no negative influence on the load balance of the task sets, and a minimum impact on the resulting *GapFactor*. However, as shown in plots 13 and 14, the reduction of costs is significant, particularly at lower laxity factors. The convergence of the costs as laxity increases can be explained by the fact that lower the laxity, the lower the initial *GapFactor*. This implies that more "useless" work is done (with a initial *GapFactor* of 100%) at lower laxities. This useless work is avoided with the adaptive initial values for *GapFactor*.

Summarizing the results of the experiments, we note that while both strategies enhance the load balance, MLS is the larger contributor. But, GFS, by introducing the idle times, improves the load balance along the time axis. Choosing a good initial value for *GapFactor* can reduce its overheads.

6 Scheduling Aperiodic Tasks

One of the motivations for the GapFactor and MaxLoad strategies, presented in the previous sections, was to increase the dynamic schedulability of aperiodic tasks.

The generation of a schedule and allocation that produces balanced loads should increase the schedulability of aperiodic tasks, because, with a well balanced allocation and schedule, all the sites of the distributed system will be equally well-placed to deal with dynamic arrivals. Additionally, the use of the *GapFactor* strategy ensures that subtasks are not clustered together, and the gap between tasks can be used to schedule aperiodic tasks. However, the discussed strategies still produce a very static schedule that defines *exactly* when each subtask should be executed and where the idle periods (gaps) are.

Here, we present two simple schemes that work on a successful static schedule for periodic tasks, produced using the algorithm described in the previous section, to provide even more leeway for the dynamic scheduling of aperiodic tasks.

The basic idea behind the schemes presented is to find earliest start times and latest start times for the "statically" scheduled periodic tasks, respecting the precedence constraints of the tasks. That is, instead of requiring that the subtasks of the periodic tasks start at their *scheduled start times*, the schemes will produce a *scheduling window* for each subtask. As long as a subtask is started within the window, feasibility of all the subtasks of a set of tasks will not be violated. This is also the requirement imposed on the schemes.

Suppose aperiodic tasks are scheduled using a dynamic task scheduler [12], [3], [14]. Given a window for each statically allocated and scheduled subtask, this scheduler will have the flexibility to decide if it is possible to delay the start of a subtask of a periodic task to run any aperiodic task that may have arrived.

The first scheme considered is called the Local Latest Start Time Scheme and the second scheme is called the Global Latest Start Time Scheme. As their names suggest, they are local and global, respectively, in their effects.

For both schemes, the scheduled start time of subtasks (generated by the static algorithm described in Section 3) is taken as the earliest start times of the tasks. This is done because that algorithm schedules the tasks at the earliest possible time, respecting precedence, placement and load balance constrains. Given the earliest start times of subtasks, tasks' deadlines and periods, the latest start time of each subtask is determined as discussed below.

To find the latest start times of subtasks, besides their logical precedence constraints, as defined in the task set specification by the designer, a physical precedence is defined between subtasks allocated to the same site. The logical precedence is part of the specification of the system and is independent of scheduling and allocation decisions made by the algorithm. The physical precedence is associated with the allocation decisions made by the algorithm. To keep the algorithms simple, the physical precedence is not altered by the local and global latest start time strategies. Altering the physical precedence will imply alterations to the allocations of subtasks on other sites.

In the Local Latest Start time scheme, the latest start times of subtasks with remote successors is the same as their earliest start time (i.e., their scheduled start time). A physical precedence relationship is established between tasks that are allocated to the same site but which do not have a logical precedence relation. The physical precedence aims at executing, at each site, the tasks in the order defined in the successful static schedule.

To compute the latest start times of subtasks, the static schedule of each site is sorted in reverse topological order and the latest start time of a subtask i (without remote successors) is defined as:

$$Latest\ start\ time_i = Min[(deadline_i), min(latest\ start\ time(successors_i))] - (computation\ time_i);$$

where $successors_i$ represents logical and physical successors of subtask i.

Given the latest start times for subtasks, the dynamic scheduler of aperiodic tasks can take local scheduling decisions that consider accepting aperiodic tasks, as long as subtasks of periodic tasks are scheduled in the predefined order, by their latest start times.

The Global scheme assumes the following. The communication front end of each site knows the order in which all messages in the system will be sent. Each message will also have a transmission window – established by the Global scheme. A message can be transmitted once all messages ahead in the order have been transmitted, and the current time is within the transmission window.

Whereas creation of scheduling windows on a site by the Local scheme did not necessitate changes to the schedules on other sites, the Global scheme will, in general, necessitate such changes. However, the order in which each site is expected to execute subtasks of a periodic task will not require modifications. This is also true of the communication channel: messages scheduled on the communication channel will be transmitted in the same order initially set by the static scheduler.

The computation of global latest start times is similar to the computation of a local latest start time, with the important difference that the communication tasks are also considered. In this case, a physical precedence is established between a subtask executing on a site and its communication with a remote successor, if any. The resulting ripple effect between sites is also taken into consideration.

At each site, we define a global latest start time at the point when a subtask is ready for execution, its immediate physical predecessor has finished execution, its earliest start time has arrived and messages from any predecessor executing on another site, if any, have arrived at the site.

The next table shows the original start times as well as the start times according to the Local and Global schemes for the schedule in figure 5. (The start times for the inter-subtask messages are not shown.) In the schedule of figure 5, two precedence related subtasks are assigned to two different sites. Hence the Local scheme is unable to alter the start times of any subtask. But the Global scheme provides considerable leeway. For example, now, subtask 4 can start any time between 73 and 89.

Table 1: Start times of subtasks

Subtask	Site	Comp Time	$Starttime_{original}$	$Starttime_{Local}$	$Starttime_{Global}$
1	2	5	0	0	0
2	3	11	33	33	33-46
3	1	7	25	25	25-47
4	2	11	73	73	73-89
5	1	7	0	0	0-18
6	2	5	29	29	29-32
7	4	6	44	44	44
8	2	7	56	56	56-66
9	3	5	68	68	68-79
10	4	6	80	80	80-91
11	1	11	37	37	37-54
12	4	11	54	54	54-66

The Local scheme does have an effect. For example, when applied to the schedule in figure 2, it will allow subtask 5 to start any time between 0 and 32, subtask 6 to start any time between 7 and 39, and subtask 7 to start any time between 12 and 44. Of course, these start times have to be obeyed in conjunction with precedence constraints.

7 Summary

In this paper, we have explored a number of ways in which the load balance characteristics of static scheduling algorithms can be improved. MLS, the Maximum Load Strategy limits the load that can be imposed on a site. GFS, the Gap factor strategy forces an idle period to follow the execution of a subtask of a task. This, in turn, forces subtasks to be more evenly distributed in a system. These two strategies, along with the two schemes for producing a scheduling interval for each task, can be used to increase the schedulability of dynamically arriving (aperiodic) tasks.

Our experiments indicate that MLS and GFS produce better balanced sites. As one would expect, these will increase the allocation and scheduling overheads. MLS has a better payoff than GFS. A proper initial values for $MaxLoad$ and $GapFactor$ will reduce these overheads. Their choice is an interesting problem in itself and deserves to be studied further.

To complete the evaluation of these strategies, we plan to study their actual effect in being able to feasibly schedule aperiodic arrivals. This will show the extent of the positive effects of increased load balance on the schedulability of dynamic arrivals.

References

[1] Bannister, J.A. and K.S. Trivedi, "Task Allocation in Fault-tolerant Distributed Systems," In *Acta Informatica*, 20, Springer-Verlag, 1983.

[2] Casavant, T.L. and J.G. Kulh, "A Taxonomy of Scheduling in General Purpose Computing Systems," *IEEE Transactions on Software Engineering*, Feb 88, pp 141-154.

[3] Cheng, S., J.A. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Real-Time Systems Symposium*, Dec 1986.

[4] Dhall, S.K. and C.L. Liu, "On a Real-Time Scheduling Problem," *Operations Research*, Vol 26, No 1, 1978, pp 127-140.

[5] Garey, M.R. and D.S. Johnson, "Strong NP-Completeness Results: Motivation, Examples, and Implications," *JACM*, Vol 25, 3, July 1978, 499-508.

[6] Krishna, C.M. and K.G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Transactions on Computers*, May 1986, pp 448-155.

[7] Lehoczky, J.P., L. Sha, and J. Strosnider, "Enhancing Aperiodic Responsiveness in a Hard Real-time Environment," *IEEE Real-Time Systems Symp.* 1987.

[8] Le Lann G., "The 802.3D Protocol: A Variation on the IEEE 802.3 Standard for Real-Time LANs," Technical Report, INRIA, July 1987.

[9] Liu, C. L. and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment," *J. ACM*, 20(1), 1973.

[10] Molesky, L.D., "Random Graph Generation in a Unix Environment," Technical Report, University of Massachusetts, Sep 1989.

[11] Peng, D.T. and K.G. Shin, "Static Allocation of Periodic Tasks with Precedence Constraints in Distributed Real-time Systems," *Proc. of the International Conference on Distributed Computing*, 190-198, June 1989.

[12] Ramamritham, K., J. Stankovic and W. Zhao, "Distributed Scheduling of Tasks With Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, August 1989, pp. 1110-1123.

[13] Ramamritham, K., "Channel Characteristics in Local Area Hard Real-Time Systems," *ISDN and Computer Networks*, 1987.

[14] Ramamritham, K., J. Stankovic and P. Shiah, "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 184-194.

[15] Ramamritham, K., "Allocation and Scheduling of Complex Periodic Tasks," *International Conference on Distributed Computing Systems*, May-June, 1990.

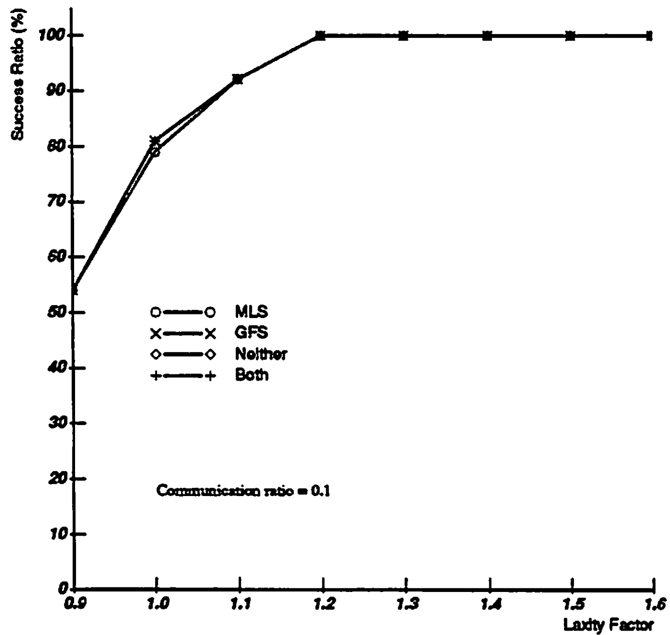
[16] Sha, L., J.P. Lehoczky and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *IEEE Real-Time Systems Symposium*, 1986.

[17] Sha, L., R. Rajkumar and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *Technical Report CMU-CS-87-181* CMU 1987.

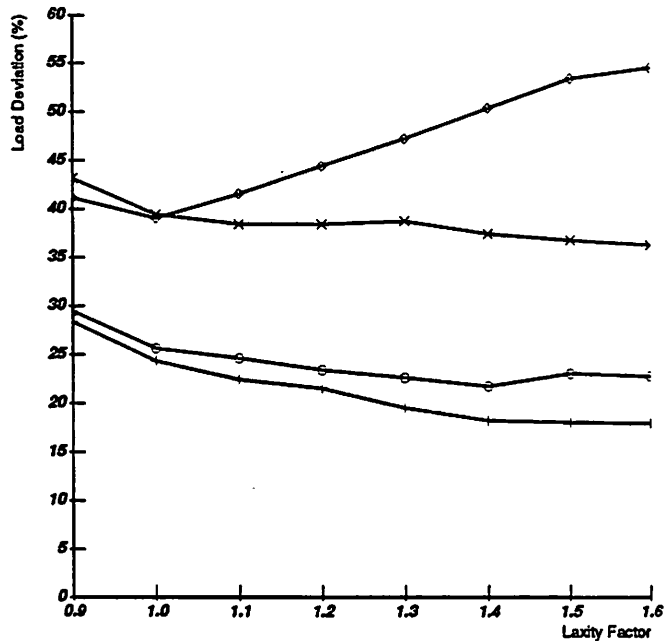
[18] Stankovic, J.A. and K. Ramamritham, *Hard Real-Time Systems*, Tutorial Text, IEEE Press, 1988.

[19] Zhao, W., K. Ramamritham and J. A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, May 1987.

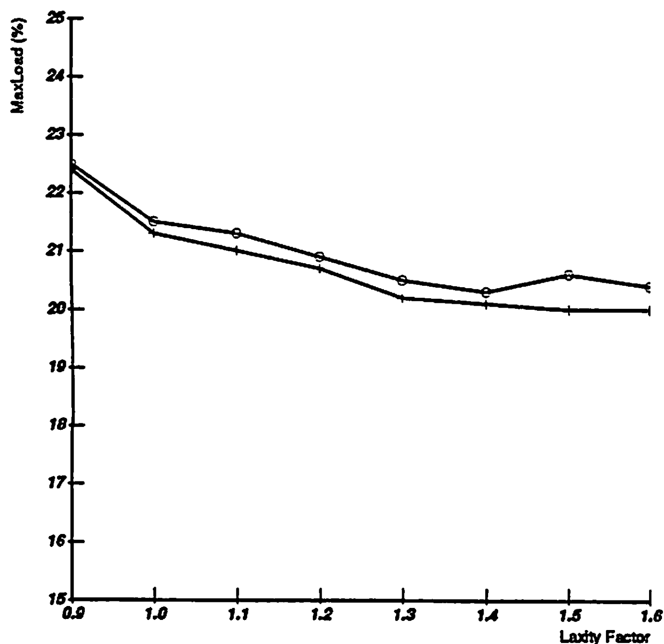
[20] Zhao, W. and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints," *Journal of Systems and Software*, 1987.



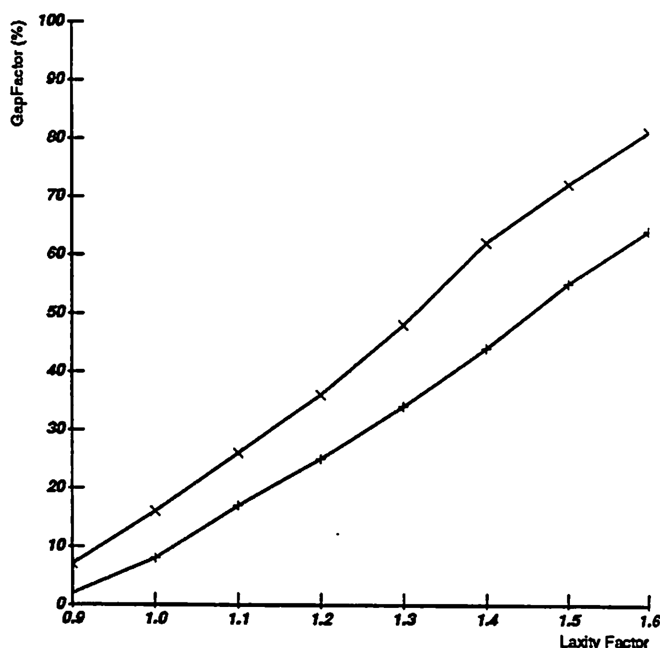
Plot1: Effect of load balancing strategies on schedulability



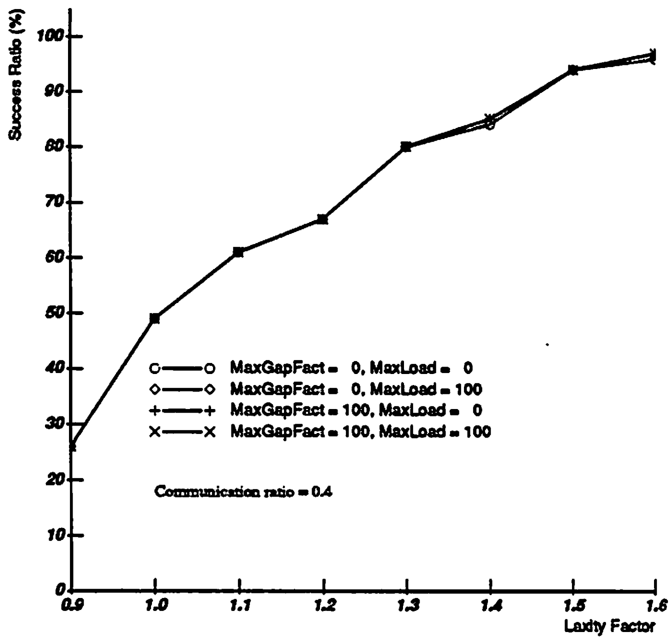
Plot2: Load balance under different strategies



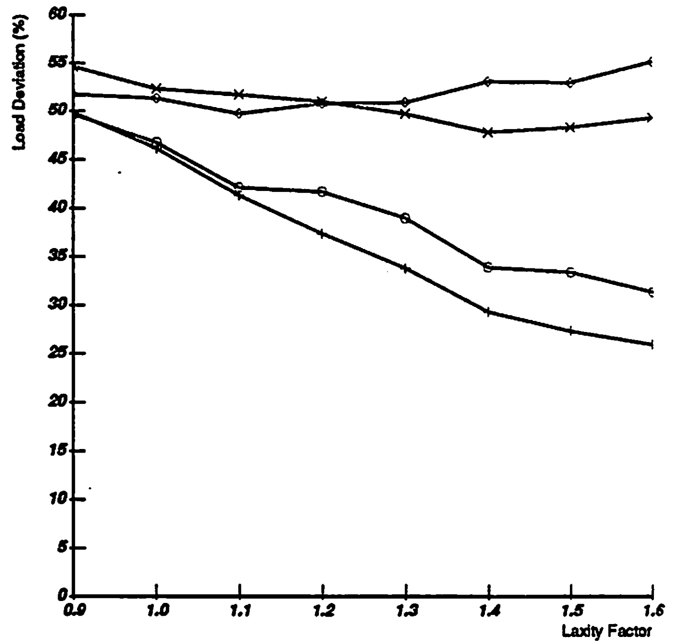
Plot3: MaxLoad for scheduled task sets



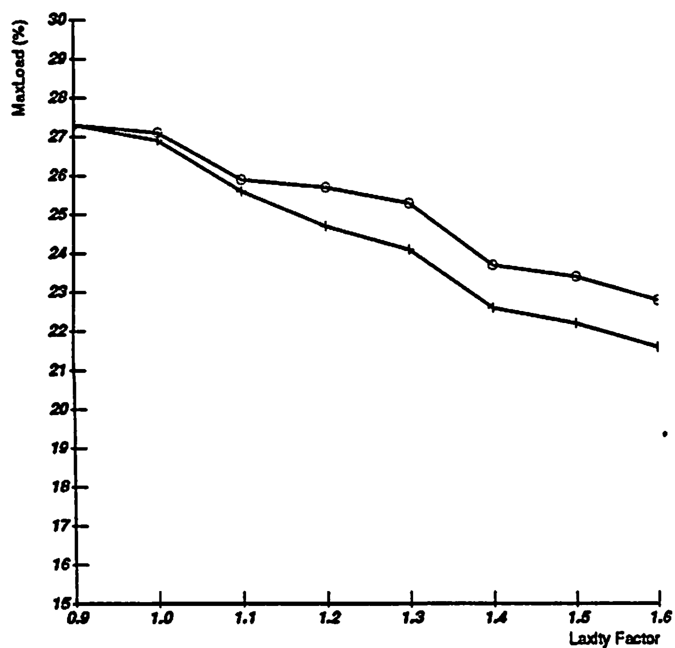
Plot4: GapFactor for scheduled task sets



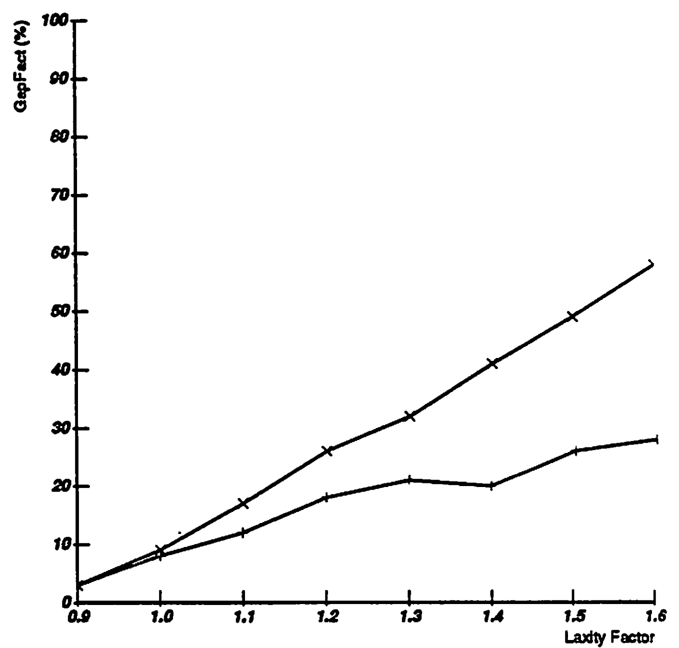
Plot5: Effect of load balancing strategies on schedulability



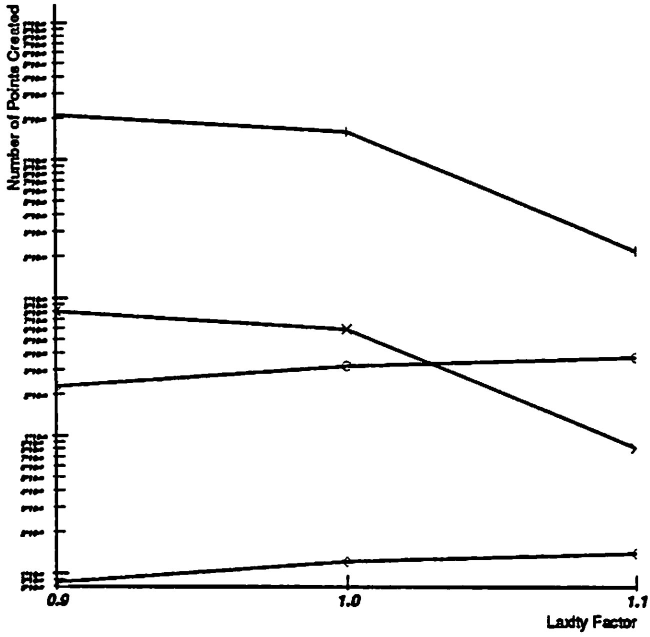
Plot6: Load balance under different strategies



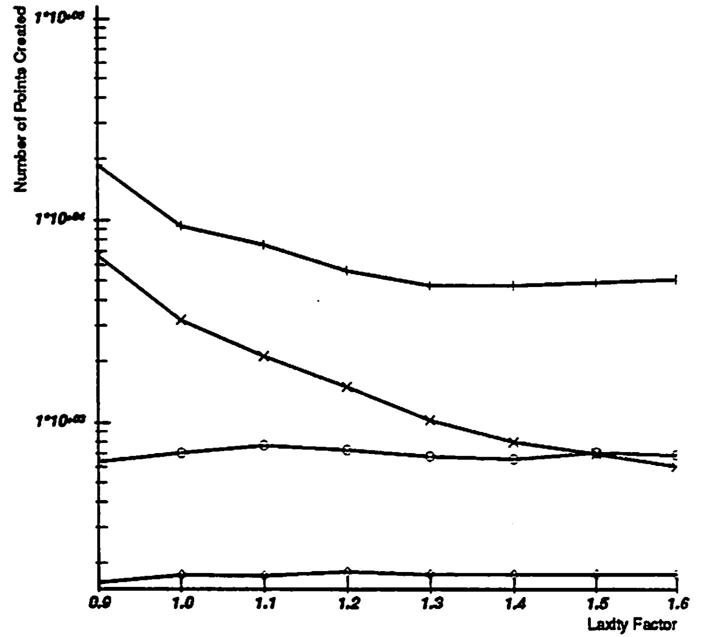
Plot7: MaxLoad for scheduled task sets



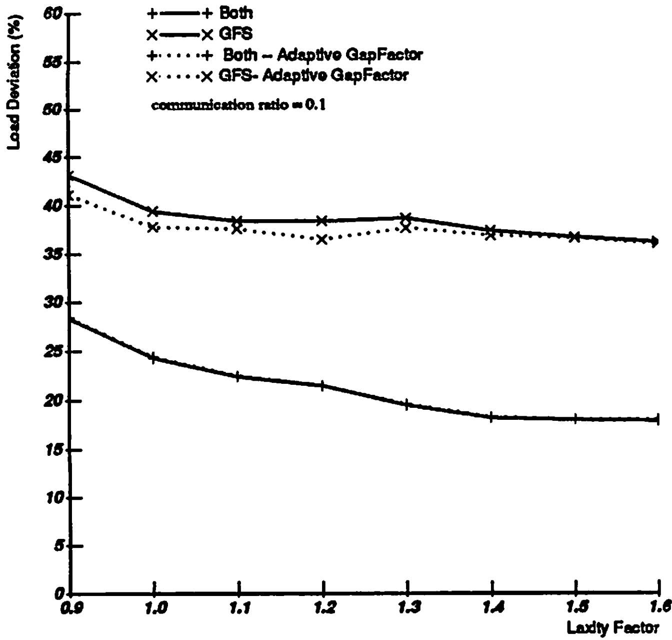
Plot8: GapFactor for scheduled taksk sets



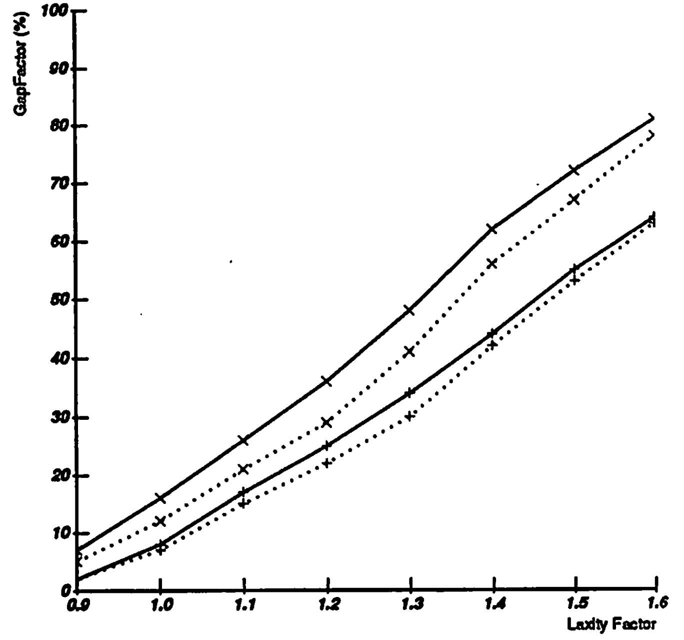
Plot9: Cost of load balancing - failure cases



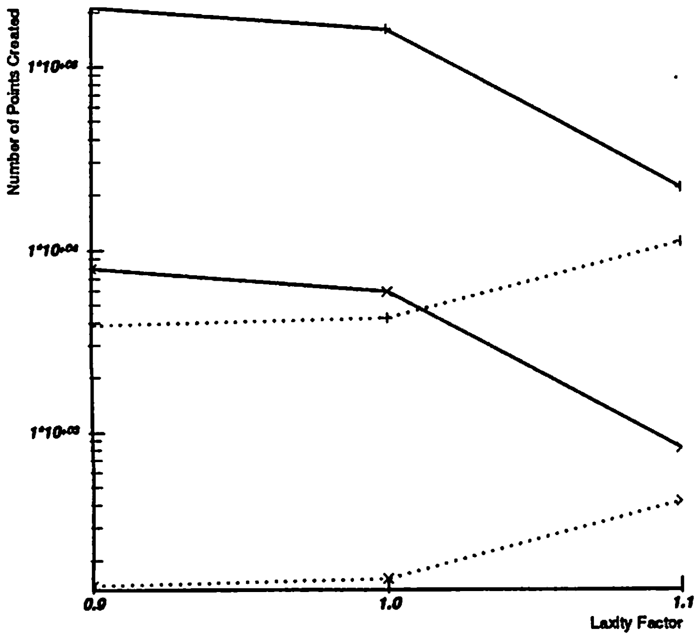
Plot10: Cost of load balancing - success cases



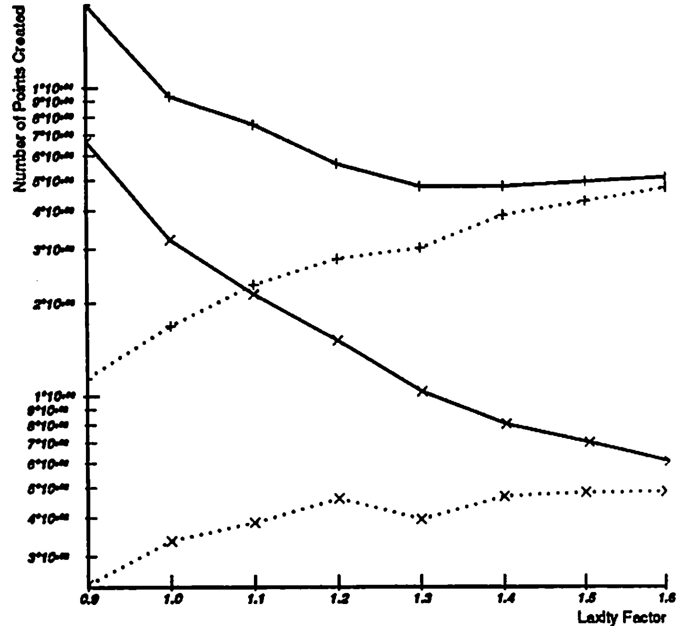
Plot11: Adaptive Setting of GapFactor



Plot12: GapFactor for scheduled taks sets



Plot 13: Cost of load balancing - failure cases



Plot14: Cost of load balancing - success cases