ALLOCATION AND SCHEDULING OF COMPLEX PERIODIC TASKS

Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-01

October 24, 1989

ALLOCATION AND SCHEDULING OF COMPLEX PERIODIC TASKS *

Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst MA 01003

October 24, 1989

ABSTRACT

Timing constraints for tasks in real-time systems can be arbitrarily complicated but the most common instances of tasks are *periodic*. This paper discusses a static algorithm for allocating and scheduling components of complex periodic tasks across sites in distributed systems. Besides dealing with the *periodicity constraints*, (which have been the sole concern of many previous algorithms), this algorithm handles *precedence*, *communication*, as well as *fault tolerance* requirements of subtasks of the tasks. As mentioned at the outset, the algorithm determines the allocation of subtasks of periodic tasks to sites, the scheduled start times of subtasks allocated to a site, and the schedule for communication along the communication channel(s). Efficacy of the algorithm is evaluated by applying it to tasks with different characteristics. The results show that the heuristics and search techniques incorporated in the algorithm are extremely effective. Specifically, they show that if a task set can be feasibly allocated and scheduled, the algorithm is highly likely to find it without any backtracking during the search.

*This work is part of the Spring Project at the University of Massachusetts and was supported in part by the National Science Foundation under grant DCR-8500332, by the Office of Naval Research under contract N00014-85-K-0398, and by a visiting fellowship from Science and Engineering Research Council (U.K.)

1 Introduction

Tasks in time-critical applications are typically categorized as being *safety-critical*, *essential* and *nonessential*. Safety-critical tasks must meet their deadlines under all circumstances, otherwise the result could be catastrophic. On the other hand, if essential tasks miss their deadlines, the performance of the system will seriously degrade. Missing the deadlines of nonessential tasks will not affect the system in the near future but may have a long term affect. Maintenance and bookkeeping activities fall in this category.

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. The algorithm described in this paper is most suitable for the static allocation and scheduling of complex, safety-critical periodic tasks. It is a static algorithm in that the decisions concerning the allocation and scheduling of components of a complex task across sites in a distributed system as well as the scheduling of communication among these components are made prior to the beginning of system execution. Besides *periodicity constraints*, tasks handled by the algorithm can have *resource requirements* and can possess *precedence*, *communication*, as well as *fault tolerance constraints*. Such tasks occur in current applications such as robotics [7] but are of special significance for next generation real-time systems, such as the space station, automated factories, and advanced command and control systems [20].

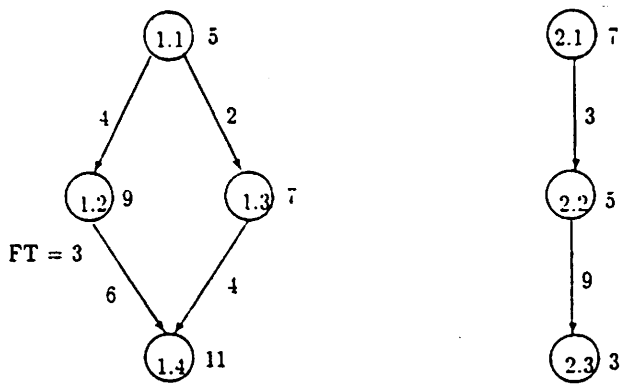
The motivation for considering complex tasks is that once there is a way to deal with tasks having complex structures and constraints, real-time systems can be conveniently programmed as communicating modules, each with its own specifications. Communication between modules can take place via shared resources or via communication ports. The modules may have producer-consumer relationships and/or may synchronize their activities. A complex program with communicating modules can be translated into a *task* composed of a set of communicating *subtasks* where each subtask involves the execution of *sequential* code and has communication, precedence, and resource constraints with other subtasks. Such subtasks can be scheduled using algorithms such as the one discussed here. Also, a complex task, for example, one requiring access to many resources, can now be (correctly) handled by breaking it up into multiple subtasks related by precedence constraints where each subtask requires a subset of the resources. While this encourages the construction of properly structured real-time programs, it should also enhance resource utilization.

Suppose we have a program with four modules, A, B, C, and D. A gets sensory information from the environment, preprocesses it and sends part of the processed information to B and the rest to C. B and C further process the information sent to them and send the results to D which displays the results. Since the processing done by B is crucial to the functioning of the system, B is required to be fault-tolerant. This set of four modules can be translated into a task composed of four subtasks with precedence, communication and fault-tolerance constraints as shown by periodic task 1 of Figure 1. Note that we have assumed that each subtask executes sequential code and hence when we refer to the communication between two subtasks, we mean the communication that occurs when one completes execution and sends its results to the other before the latter begins execution. Periodic task 2 of Figure 1 is another example of a task with three subtasks related by simple precedence.

The algorithm consists of two parts. The first part decides whether a cluster of communicating subtasks of a task should be assigned to the same site. This decision is based on the computation times of the subtasks in a cluster and the amount of communication between them. This part is heuristic in nature and we compare the performance of different heuristics. Given the clustering done in the first part, the second part assigns the clusters of subtasks to the sites in a system and also determines a feasible schedule, if possible, for the subtasks as well as the communication between them. This is done using a search driven by task characteristics, where, at each point in the search, subtasks eligible for execution are considered in accordance with task characteristics such as latest-start-times and precedence constraints. Since the first part of the algorithm eliminates some of the communication (by deciding that certain subtasks should be assigned to the same site), the search space in the second part is considerably reduced.

A majority of current scheduling work dealing with periodic tasks assume that tasks have only periodicity constraints [10] [1], [17]. Constraints such as those that arise from resource sharing and synchronization are beginning to be considered [18]. Resource constrained scheduling is also investigated in [21] [22] but they deal with periodic tasks that execute on a single site in a distributed system. The work described in [13] comes closest to ours but differs in a number of ways: Our algorithm allows subtasks of a task to execute on different nodes thereby increasing the flexibility of resource usage; They allocate all subtasks of a task to the same node. Our algorithm handles fault-tolerance constraints; They have not considered fault-tolerance issues. We assume that only subtasks within a periodic task have precedence relationships; They assume that precedence relationships can also exist between (subtasks of) different periodic tasks. Finally, for practical reasons, we use heuristics-directed search; Their's is a pure branch and bound search and its practicality is yet to be demonstrated.

The rest of the paper is structured as follows: In Section 2, the characteristics of the periodic tasks considered by the scheduling algorithm are discussed. Assumptions made about the resources in the system are also clarified. Section 3 forms the crux of the paper. Herein the details of the algorithm are provided. Results of experimental evaluation of its performance for different types of complex tasks are discussed in section 4. A number of extensions to the algorithm are presented in Section 5 including ways to schedule aperiodic tasks in the presence of periodic tasks. Section 6 concludes the paper by summarizing the important characteristics of the algorithm and discussing its applicability.



Periodic Task 1:
Subtasks = 1.1, 1.2, 1.3, and 1.4
Period = 50
Subtask 1.2 is required to be executed in triplicate; Subtask 1.4 votes on the results communicated by the three copies of 1.2.

Periodic Task 2:
Subtasks = 2.1, 2.2, and 2.3
Period = 25

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

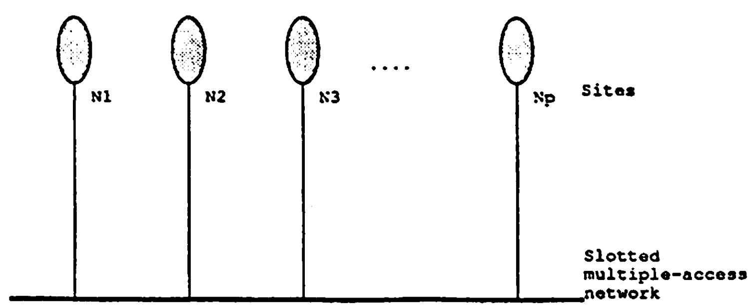


Figure 2: Schematic of a Simple Distributed System

2 Task and System Characteristics

For the purposes of the algorithm, a distributed system consists of a number of sites, with a set of resources attached to each site. The algorithm is designed to work with 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. For instance, point-to-point networks or multi-access networks employing a TDMA protocol have such predictability.

To simplify the following discussions, the assumption is made that each site in the distributed system has one processing element and a given set of (passive) resources. The sites are connected by a *slotted multiple-access network*. Figure 2 shows the schematic of the network. Communication from one site to another occurs at pre-specified times, as per the schedule generated. To be more precise, a set of consecutive slots of the communication channel is dedicated for communication between a given pair of subtasks.

Specifications of periodic tasks include the following: (See figure 1 for the specification of two sample periodic tasks):

1. The period of the task. The semantics assumed is that one instance of all subtasks of a task should be executed every period.

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

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

   It is assumed that the execution of each subtask *cannot* be preempted.

4. The maximum amount of information communicated from a subtask to its successor is expressed via a value attached to the corresponding arc. This is used to determine the communication delays incurred due to information transfer from a subtask to its successor if they are scheduled on different nodes in a distributed system. This information is also used to schedule the communication between communicating subtasks placed on different sites. (Communication within a site is assumed to incur zero delay.)

   The actual delay incurred by communication from one subtask to another depends on the amount of information transferred, the type of communication network, and the protocol used. As mentioned earlier, in this paper, communication among sites is assumed to occur along a slotted multiple-access network. Also, we assume that the value associated with arcs in the graph are the communication times for the corresponding messages. Such a view simplifies subsequent discussion. In Section 5 we discuss how more complex communication mechanisms can be accommodated.

5. The fault tolerance requirements of subtasks is specified by a value attached to each subtask indicating the number of replicates needed for the subtask. For example, if three replicates of a subtask are needed, then three instances of the subtask are scheduled on three different sites and each of them is associated with the specifications of the original subtask. (Thus, it is assumed that fault tolerance is achieved via replication. The results of the replicates of a subtask are sent to each successor of the subtask which, depending on the fault model assumed, may vote on the results to determine the valid input. If voting is done, it is assumed that the voting overheads incurred by the successor are either negligible or are already accounted for in the computation time of the successor.)

6. Resource constraints 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 for 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.

3 Constrained Search for a Feasible Schedule

The purpose of our scheduling algorithm is to allocate subtasks of a set of tasks across sites in a distributed system and to schedule the subtasks such that the tasks meet their periodicity requirements. It is well-known that even some of the simplest scheduling problems are NP-hard in the *strong* sense [4] and hence in practice it is not possible to determine optimal schedules efficiently. In this section, we discuss the practical problems encountered in scheduling complex periodic tasks and the heuristic solutions adopted in the proposed algorithm. The heuristics are necessary to constrain the search space to manageable levels.

It should be remembered in the course of the following discussion that the algorithm is a static algorithm where the scheduling entity has the necessary information about all the tasks as well as all the sites and the communication medium connecting them.

When allocating and scheduling subtasks that communicate, three issues have to be dealt with in conjunction.

1. Given a set of communicating subtasks, should they be placed on the same site?

2. Which site should a subtask be allocated to?

3. Once a subtask is allocated to a site, when should it begin execution?

An optimal solution should consider the cross product of the solution space of each of these queries. As we will presently show, this is impractical for non-trivial distributed systems and for complex periodic tasks.

Ideally, we should cluster subtasks of a task such that (1) the cost of communication among subtasks within a cluster is higher than that between subtasks in different clusters and (2) the cost of communication among subtasks within a cluster negate the advantages of the parallel execution of the subtasks at different sites. Subtasks belonging to the same cluster should be allocated to the same site. The basic idea then is to cluster together subtasks that have "substantial" amounts of communication among them. This strategy attempts to eliminate the larger communication costs.

While more elaborate clustering mechanisms are under investigation, in this paper, we present the results for clusters of size two. In this case, suppose we have n pairs of communicating subtasks. If an *optimal* solution has to be found, the 2^n different clustering possibilities that exist must be examined. In our experiments we consider task sets that have over 70 subtasks with around half as many pairs of communicating subtasks. Even assuming it takes one μsec to examine each possibility, it will take *more than a day* to exhaust all of them. The complexity of finding an optimal solution poses a further practical problem when we determine *where* and *when* each subtask in a cluster should be scheduled to execute. Hence we approach each of these issues using heuristics. Our experiments show that the adopted heuristics are extremely effective. Specifically, they show that if a task set can be feasibly allocated and scheduled, very likely, the algorithm will find it without any backtracking during the search.

We first give a broad overview of the algorithm. Details needed to gain a complete understanding of the algorithm as well as the rationale for the approach follow.

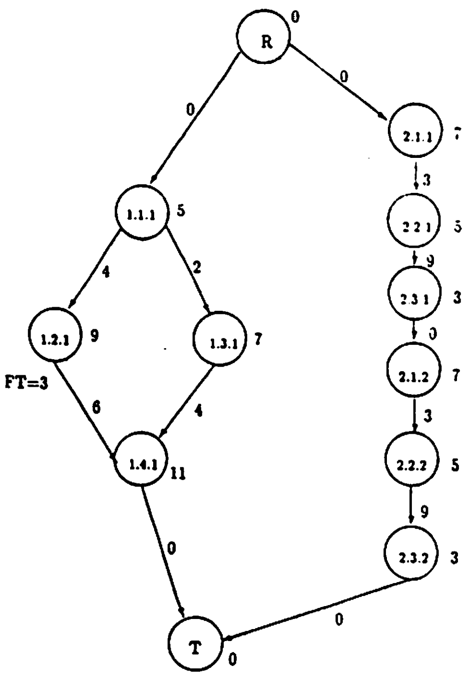
Overview of the Algorithm

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.

Given the graph depicting each task, Step-I constructs the *comprehensive graph* containing all instances of the tasks that will execute in an interval of length L. Figure 3 shows the comprehensive graph given a system with the two tasks shown in Figure 1.

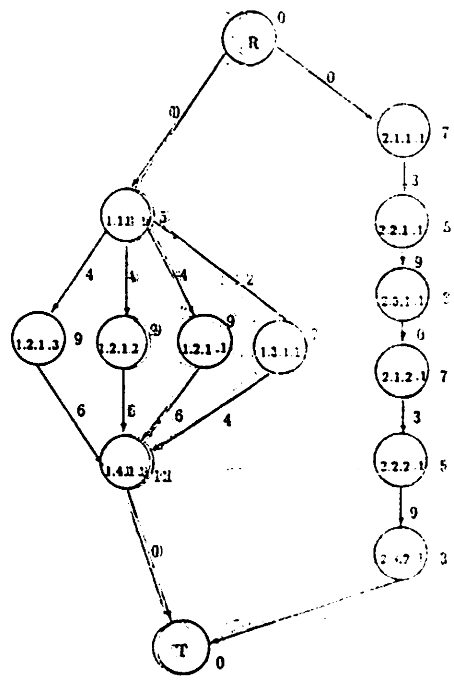
In Step-II, the comprehensive graph is embellished with replicates of the subtasks that have fault-tolerance requirements. This results in the addition of some subtasks and some arcs to the graph. Figure 4 shows the graph that results after including the fault tolerance requirements of subtask 1.2.

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 *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 gen-



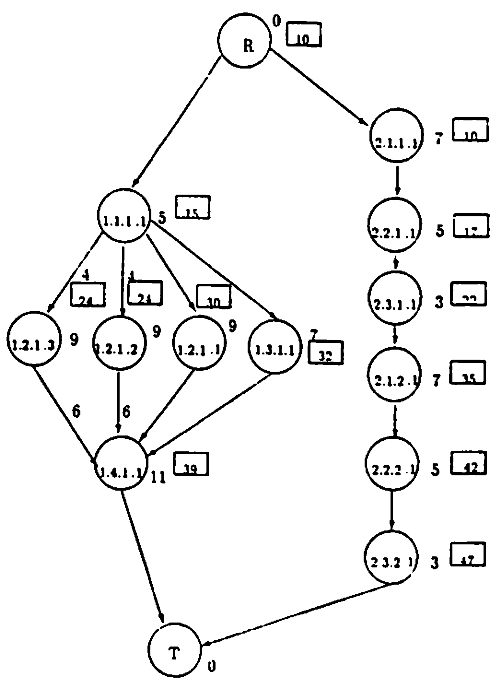
Here the k^{th} instance of the j^{th} subtask of periodic task i is identified by i.j.k. 2.3.1 has a deadline of 25. 2.1.2 has a start time constraint of 25, i.e., can not start executing before 25. 1.4.1 and 2.3.2 have a deadline of 50.

Figure 3: Comprehensive Graph



Only subtask 1.2.1 requires replication.
The additional suffix added to the identity of a subtask indicates the number of the replicate

Figure 4: Comprehensive Graph Considering Fault Tolerance constraints



The subtasks have start time constraints and deadlines as shown in Figure 3. The rectangle next to each subtask indicates the latest start time of the subtask. Subtasks connected by an arc with no numbers attached to them should be assigned to the same site. Where such numbers are attached, they denote communication costs.

Figure 5: Communication Graph

erated with the current value of CF. Figure 5 is the communication graph generated from the graph of Figure 4 when the value of CF is set to eliminate all communication overheads (except to and from replicates of subtasks).

Step-IV allocates the subtasks to sites in the system, allocates the communication (between subtasks) to the 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 conjunction. 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. Figure 7 depicts the allocation and scheduling decisions made for the subtasks in Figure 5.

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 the following sections, we provide details of each of these steps.

3.1 Construction of the Comprehensive graph

The following semantics is associated with periodic tasks: All subtasks of the j^{th} instance of a periodic task with period P should be completed between $(P \times (j-1))$ and $(P \times j)$ (of course, respecting the various constraints associated with the subtasks). Given these semantics, the algorithm attempts to construct a feasible schedule for all task instances that should execute within the interval $(0, L)$ where L is the Least Common Multiple of the periods of all the periodic tasks involved[1]. The first instance of each periodic task is ready to begin execution at time 0.

The *comprehensive graph* is composed of

$$N_i = \frac{L}{P_i} \tag{1}$$

instances of the i^{th} periodic task (with period P_i). In addition to the other constraints, the j^{th} instance of task i, for $j = 1, 2, \ldots, N_i$, will have a *start time constraint*: Execution of the task, specifically, its first subtask(s) (i.e., those that have no predecessors), can not be scheduled to begin before $((j-1) \times P_i)$. Also, the last subtask(s) of the the j^{th} instance of task i, (i.e., those which have no successors), will have a deadline of $(j \times P_i)$.

Figure 4 shows the comprehensive graph constructed for the tasks in figure 3. Note the presence of a dummy root node (R) and a dummy terminal node (T) in the graph.

[1]This will be inappropriate when, for example, the periods are relatively prime. But, in practice, periods can be adjusted to produce L values that are manageable.

3.2 Dealing with Fault-tolerant Subtasks

Because of space limitations we do not discuss the applicability of different fault tolerance techniques to real-time systems. For the purposes of this paper, we assume that that not all subtasks in a complex task need be replicated. This is based on the observation that fault masking and forward error recovery techniques can coexist without difficulty: Resources whose consistency is required in spite of failures are replicated and subtasks that access such resources are also replicated. Similarly, subtasks whose execution should be completed on time in spite of failures need be replicated. Thus, for such resources and subtasks, faults are masked through replication. Resources whose states can be approximated (for example, from recent *recovery point(s)* or by assuming a default/average value) following a failure need not be replicated. Subtasks accessing such resources also need not be replicated. Similarly, subtasks whose outputs can be approximated by their successors, in case the subtask fails, need not be replicated. (Quite often, depending on the the nature of the resources, a resource need not be completely replicated. We do not consider such cases here.)

In the periodic tasks of Figure 3, only 1.2.1 needs to be replicated. Figure 4 shows the effect of considering fault-tolerance requirements. Note that the additional replicates of a subtask added to the graph are endowed with the same specifications as the original subtask.

3.3 Constructing the Communication Graph

We chose the following heuristic approach to address the issue of whether a set of communicating subtasks should be allocated to the same site: Determine whether or not two subtasks should be placed on the same site on the basis of their computation times and the amount of communication between them. There is a tradeoff here and it depends on the communication costs and computation time of the subtasks. If the two subtasks are placed on the same site, the costs of communication between the two subtasks are avoided and hence the time required to complete this pair of subtasks is reduced. However, the load on the site to which both subtasks are allocated increases which in turn can prevent it from taking on a subtask that can potentially execute in parallel. In any case, suppose we had two pairs of subtasks where the computational characteristics of one pair are the same as the other. Then, intuitively, it is better to assign subtasks with the higher communication costs to the same site. This is the basis for the heuristic underlying our clustering scheme.

Specifically, two subtasks with computation times C_i and C_j where the communication from the first subtask to the second takes $Comm_{ij}$ units of time will be placed on the *same site* if the following holds:

$$(C_i + C_j) < (CF \times Comm_{ij}) \tag{2}$$

CF stands for a tunable parameter called *communication factor*. For a given value of CF, this scheme tends to assign the pair of subtasks with higher communication costs to the same site. In our algorithm, if a feasible schedule is not arrived at with a

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

certain value of CF, the scheduling algorithm reattempts after adjusting the value of CF. It should be clear that the maximum value of CF that needs to be considered is

$$max \ \ \forall \, i,j, \ \frac{(C_i + C_j)}{Comm_{ij}} + \epsilon. \tag{3}$$

for all communicating subtasks i and j. Let us call this $maxcf$. (ϵ has some positive non-zero value. It is set to 1 in the experiments of Section 4.)

Assume that $Comm_{ij}$ for every pair of communicating subtasks i and j is non-zero. This is true in practice since completion of a subtask's execution has to be notified to the successor, say via an "enabling signal". Thus, assigning a value of $maxcf$ to CF will force all communicating subtasks to be allocated to the same site. More and more pairs of communicating subtasks will be separated as the value of CF is decreased from $maxcf$. A CF value of 0 will force communicating subtasks to be allocated to different sites. By making $Comm_{ij}$ infinity (zero) between a pair of subtasks i and j (even if it is not) the subtasks involved can be forced to be be allocated to the same (different) site.

Given a comprehensive graph, the above "pairwise heuristic" is applied to every pair of communicating subtasks to determine whether the subtasks should be assigned to the same site. The resulting graph is the *communication graph*. Figure 5 shows the *communication graph* of the graph in Figure 4 for $CF = maxcf$. Subtasks which are connected by arcs that do not have any associated times are the ones that must be allocated to the same site. In this case, since we are considering $CF = maxcf$, two subtasks that communicate are assigned to the same site. When applied to every pair of communicating subtasks, this scheme eliminates all communication. Clearly, communication to and from the replicates of a subtask (for example replicates of 1.2.1 in figure 3) can not be eliminated since by definition these replicates have to execute on multiple nodes.

In Section 4, we evaluate the performance of the above heuristic and compare it with a "random" scheme where whether or not to retain a communication link is decided by the toss of a coin.

In general, when two subtasks communicate, they incur overheads for packaging and unpackaging the messages involved in the communication. These overheads can be accounted for by adding them to the computation times of the subtasks allocated to different sites. However, in the rest of the paper, we assume that these are negligible and hence do not alter the computation times of the subtasks.

Once the *communication graph* has been derived, we determine the *latest start time* of each subtask s in the *communication graph*. Assume s is a subtask of periodic task (instance) T with deadline D. Let us define the *length of a path* between two subtasks in the *communication graph* to be the sum of the computation times of all the subtasks, including the subtasks in consideration, plus the sum of the communication times, if any, associated with the arcs that lie along the path. Let LP be the length of the longest path from s up to and including the last subtask of T. The latest start time of s is defined to be $D - LP$. Latest start times of tasks are used in ordering tasks for consideration during scheduling.

To ease further discussions, we refer to the communications that have to be scheduled as *communication subtasks* and the subtasks that must be allocated to sites as *CPU subtasks.* This nomenclature recognizes the additional resource constraint imposed by subtasks: Whereas CPU subtasks are allocated and scheduled on (processing) sites, communication subtasks are scheduled on the (communication) network. Viewing it in this fashion allows us to uniformly deal with all (types of) subtasks, given that the algorithm takes resource constraints into account.

3.4 Making Allocation and Scheduling Decisions

At this point in the algorithm, we have a set of CPU and communication subtasks related by precedence constraints. The communication graph depicts this relationship. Note that so far the algorithm has only determined which subtasks should be assigned to the same site, but without determining *the* site. We first outline the scheme.

A subtask becomes *enabled* only when all its predecessors have completed execution. Thus, at any given time, only some of the subtasks are eligible for consideration. A subtask becomes *ready* only if it is enabled and when its start-time constraint is met. Given a list of ready subtasks, the order in which we consider them for allocation and scheduling will determine whether or not a feasible schedule is derived. For example, assume that at a given site two subtasks are enabled at time 20. Each subtask has computation time 10; The first has a deadline of 30 and the second 40. In this case, if we do not consider the subtasks according to their latest start times or deadlines, a feasible schedule can not be arrived at. In general, delaying the execution of the subtask with the least latest start time will delay the overall completion time of the set of tasks. Hence, the ready list is ordered according to increasing latest start time of the subtasks in the list. If there is a tie, the subtask with more number of successors is placed first. This is equivalent to using the LST/MISF (Latest Start Time/Maximum Immediate Successors First) heuristic during the search for a feasible allocation and schedule.

Since we are assuming that each site has only one processor, when the processor is busy (idle), the site is busy (idle). Some sites may be busy executing previously scheduled CPU subtasks that are yet to finish. Clearly, since CPU subtasks are assumed to be non-preemptable, ready subtasks can be scheduled only on currently idle sites. For uniformity, we do not allow communication subtasks to be preempted either. Thus, a communication subtask can be scheduled only if the communication channel is idle. When making allocation and scheduling decisions, we refer to an idle site or an idle communication channel as a *schedulable resource.*

After initializing the ready list with the root node of the communication graph, search proceeds as follows. At each search point, the algorithm first checks if the allocation and scheduling decisions made thus far will not lead to a feasible schedule. These checks are discussed under "Testing for Infeasibility".

- If the checks indicate that a feasible schedule is likely, then the subtasks in the ready list are *mapped* to schedulable resources. Obviously, there are a number of possible mappings and they are generated and considered in order (as discussed under "Systematic Generation of Mappings").

- If the current mapping is *valid*, i.e., meets certain requirements (discussed under "Testing the Validity of a Mapping), the search path is extended by one more level and the search proceeds. The "time" corresponding to the new level is set to be the smaller of min(earliest start time of currently *enabled* tasks) and min(earliest completion time of subtasks currently occupying resources). Subtasks that were in the previous ready list but not scheduled are placed in the new ready list. Tasks that have just become ready are added to the new ready list.
- If the current mapping is *invalid*, the next mapping is generated and its validity determined. If no more valid mappings exist at the current point of search the algorithm discards the current search point. Once this occurs, if the algorithm is allowed to backtrack, it backtracks to the previous search point. On going back to the previous search point, the next valid mapping, if any, at that point is pursued. In Section 4 we test the algorithm's performance with and without backtracking.

- If it is found that the current set of allocation and scheduling decisions will not lead to a feasible schedule, the current search point is "bound", i.e., is discarded. Here again, if backtracking is allowed, the algorithm backtracks to the previous search point and proceeds (if possible) with the next valid mapping at that point.

Experimental results show that the LST/MISF based ordering of the ready list works effectively in conjunction with the systematic generation of mappings, the tests used to validate a given mapping, and the checks used to determine whether the current search path will lead to a feasible schedule. Hence we discuss these now.

Systematic Generation of Mappings: Given subtasks in the ready list, a mapping defines the assignment of subtasks to a schedulable resource. To simplify the generation of mappings, we introduce the notion of "idle subtasks". Sometimes, resources may remain idle because none of the ready tasks require it. Also, because of the characteristics of subtasks that become enabled or become ready at a future time, at times, it may be better to allow a resource to remain idle even if a currently ready task can be scheduled on it. For instance, a task that is yet to become ready may have an earlier latest start time than another that is ready. Further, if the resource requirements of the former conflict with the latter, then immediately scheduling the latter may affect the schedulability of the former. Thus, we have to consider "scheduling" not only the subtasks, but also the *idle subtasks*. *Idle* subtasks represent time slots during which one or more schedulable resources are allowed to remain idle. The notion of idle subtasks allows us to treat resource assignment uniformly: *Some* subtask is always assigned to a schedulable resource; if it happens to be the idle subtask, the resource remains idle. To facilitate this scheme, a number of idle subtasks, equal to the number of sites idle at this point, are appended to the ready list.

Suppose there are n subtasks (excluding idle tasks) in the ready list and k idle schedulable resources at a certain point in the search. Considering idle subtasks, the effective size of the ready list is $(n + k)$. There are $O((n + k)^k)$ possible mappings from subtasks to schedulable resources, considering idle subtasks as well. A mapping

$(m_1, m_2, ...m_k)$ represents the assignment of the m_i^{th} task in the ready list to the i^{th} idle schedulable resource. If the possible mappings from subtasks to schedulable resources are generated systematically, then given a certain ready list and a particular mapping, the next possible mapping can be determined without any other information. This scheme, inspired by the one used in [6], considerably reduces the amount of information that has to be maintained during search: only the most recently used mapping at this point in the search needs to be kept as part of the search structure.

For example, suppose subtasks $t1$ and $t2$ are ready, $t1$ has a lower latest start time, and schedulable resources $s1$ and $s2$ are available. Then $n = 2$ and $k = 2$. The mappings from the *ready list* $(t1, t2, idle, idle)$ to the *list of idle schedulable resources* $(s1, s2)$ will be generated in the following sequence:

$$(1,2), (1,3), (2,1), (2,3), (3,1), (3,2), (3,4). \tag{4}$$

This is a lexicographically ordered sequence where each mapping has k elements and each element in a mapping is between 1 and $n+k$. Mappings that have the same effect as a previously generated mapping are not generated. The above sequence corresponds to the following sequence of mappings:

$$(t1, t2), (t1, idle), (t2, t1), (t2, idle), (idle, t1), (idle, t2), (idle, idle). \tag{5}$$

Observe that mapping (2,4) (corresponding to (t2, idle)) is not generated since it has the same effect as (2,3). For similar reasons, (4,1), (4,2), and (4,3) are also not generated.

Testing the Validity of a Mapping: The following conditions will have to be met for a mapping to be valid.

- As noted earlier, schedulable resources can be allowed to remain idle. However,
  - when there are ready tasks, not all schedulable resources in the system can remain idle.
  - if the subtask with the lowest latest start time among subtasks yet to be scheduled is ready and is schedulable, it has to be scheduled.
- Resource constraints must be met:
  - When a CPU subtask is mapped to a site, the resources needed by the subtask should be available in that site. Communication subtasks can be allocated only to the communication channels.
  - If two subtasks are separated by a communication subtask, the two subtasks should be scheduled on different sites.
  - Replicates of a subtask (if any) should be scheduled on different sites.
  - Subtasks t' and t'' should be assigned to the same site if they have a CPU subtask t as their common successor.

Testing for Infeasibility: The current search point will not lead to a feasible schedule, i.e., a schedule that meets all the timing constraints, if one of the following hold. First, *time* at the current search point is greater than the latest start time of a task in the ready list. Second, the total time available on a particular resource between the current time and L is less than that required by the subtasks that will execute between now and L and require that resource. For example, assume that communication is via a multiple access network and the current time is t. If the time needed for all the communication subtasks that have not yet been scheduled at this point is greater than $(L - t)$, then the current search will not lead to a feasible schedule.

In addition to these two cases, by "looking ahead", a potentially infeasible schedule can be detected sooner. Consider periodic task 1 of Figure 1 without the fault tolerance requirement on subtask 1.2. Assume that all the communication subtasks of this task have been eliminated, i.e., all the CPU subtasks must be scheduled on the same site.

Suppose subtasks 1.2 and 1.3 are currently on the ready list. Then the following condition should hold:

$$(current\ time + C_{1.2} + C_{1.3} <= LST_{1.4}) \qquad (6)$$

where C_i is the computation time of subtask i and LST_i is the latest start time of i. The reason for the above conditions should be obvious: Since both subtasks must execute on one site, they should complete execution before the latest start time of the successor task. If the above condition does not hold, the current partial schedule will not lead to a feasible schedule.

Suppose instead that subtask 1.1 is on the ready list. Then the following condition should be satisfied for the search to proceed:

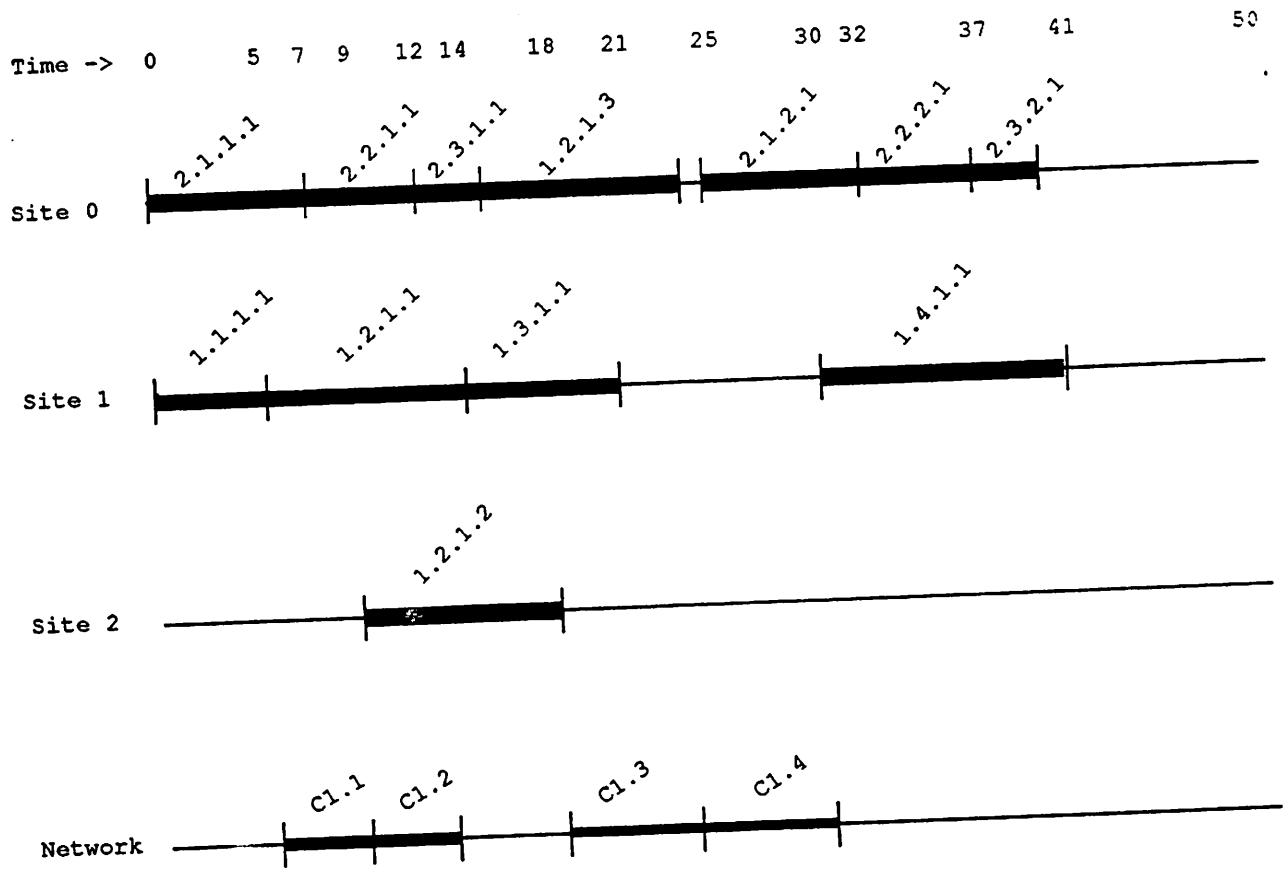
$$(current\ time + C_{1.1} + C_{1.2} + C_{1.3} <= LST_{1.4}) \qquad (7)$$

Clearly, the above specific cases can be generalized. However, overheads are involved in detecting whether the conditions for the "look ahead" apply and hence all our experiments (see Section 4) exploit just these simple situations. In general, the sooner we determine that the search path being pursued will not lead to a feasible schedule the less time and resources will be wasted on scheduling. That is, the more search points we are able to "bound" and the more mappings we are able to identify as being invalid, the faster we can determine a feasible schedule. Thus, it is important to constrain the search as much as possible.

Our experiments indicate that the LST/MISF heuristic in conjunction with the mapping generation and validation scheme as well as the search path bounding scheme determine the initial 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. In any case, the algorithm produces monotonically better solutions as the number of allowed backtracks is increased. Specifically, the percentage of task sets for which feasible solutions can be found either stays the same or displays slight improvement with an increase in the number of backtracks.

```
procedure search-for-a-feasible-schedule;
{int CF;
 set the current value of CF;
 construct communication graph;
 //initialize first search point
 time = 0;
 move root node of communication graph to the ready list of current point;
 while <ready list is not empty>
      if (<current point will not lead to a feasible schedule>
          OR
          <no (more) valid mappings exist at this search point>)
            {discard information about the point;
             if < previous point exists>
                backtrack to previous point;
             else
                exit;
            }
      else
         {schedule tasks from ready list according to next
            valid mapping;
          //begin consideration of next search point
          time corresponding to next point
            =
          smaller(min(earliest start time of enabled tasks),
                  min(earliest completion time of subtasks
                        currently occupying resources));
          move subtasks that were ready but not scheduled at
            the previous point to new ready list;
          move tasks that have just become ready to ready list;
          fill ready list with appropriate number of idle subtasks;
         }
}
```

Figure 6: Pseudo Code for Scheduling Subtasks in a Communication Graph



C1.1 represents communication between 1.1.1.1 and 1.2.1.2.

C1.2 represents communication between 1.1.1.1 and 1.2.1.3.

C1.3 represents communication between 1.2.1.2 and 1.4.1.1.

C1.4 represents communication between 1.2.1.3 and 1.4.1.1.

Figure 7: Assignment and Schedule for Two Periodic Tasks

Figure 6 presents the pseudo code for scheduling subtasks in a *communication graph*.

Figure 7 shows the schedule that results when the algorithm is applied to the *communication graph* of Figure 5. It shows the system resource to which each subtask is allocated along with the scheduled start time of the subtask. Here all subtasks of Task 1 execute on site 1, all subtasks of Task 2 execute on site 0, the redundant copies of subtask 1.2.1.1 execute on sites 0 and 2. Derivation of this schedule required no backtracking.

4 Experimental Evaluation of the Algorithm

We have implemented the above algorithm and have tested it with many complex periodic tasks. The progression of graphs starting from figure 4 and culminating in the graph of figure 6 as well as the schedule in figure 7 were generated by this implementation.

In this section, we first discuss the experimental setup and then present results of various experiments. The algorithm is implemented in C++ and consists of two basic components: The generation of a *communication graph* and the making of allocation and scheduling decisions.

Whereas we did extensive experimentation with the algorithm under various parameter settings and different task types, due to space limitations, here we show only some of the salient results. Observations that can be made from these test cases were corroborated by results not reported here.

Each point in the graphs depicted in plots 1.1 through 5.6 is produced by testing the performance of our algorithm (as well as the ones with which it was compared) on 100 different periodic task sets. The precedence structure of each periodic task was generated using the random graph generation package discussed in [12]. 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: (Some experiments were done with graphs having slightly different properties; the differences are indicated at the pertinent points in the section.)

- 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:

$$P = C \times task\_size \times (1 + (redundancy\_ratio \times redundancy\_no)) \times laxity\_factor \quad (8)$$

Note that this formula is based on just the computational requirements of the tasks. The computational requirements of a task increases 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_facto., the larger the period of the task.

- 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 nodes and depending on the communication factor (CF), comm_ratio, redundancy_ratio, and the redundancy_no, the communication graphs had between 100 and 150 nodes.

Even though we generate 100 task sets to obtain each point in the graphs, we removed from consideration task sets that were definitely infeasible. This was detected by determining the latest start time of the root subtask of a periodic task, ignoring all communication. If the latest start time 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.) Obviously, this does not eliminate all infeasible task sets since the presence of communication costs can make some task sets infeasible. However, as mentioned earlier, the problem of determining if a given set of periodic tasks is feasibly schedulable is a computationally intractable problem. Because of this, when a heuristic algorithm does not succeed in determining a feasible schedule, it could be due to the infeasibility of the task set. Still, if one heuristic scheme or one combination of parameter settings for an algorithm is able to feasibly schedule a periodic task set while another is not, we can clearly conclude the superiority of the first. That is one means we adopt in our experimentation to compare competing schemes. Further, we tried to obtain evidence to indicate that when our algorithm fails in determining a feasible schedule, the task set is highly likely to be infeasible. We point out such evidence in the course of the following sections.

The configuration of the distributed system assumed is shown in figure 2. The tests involved a system with *six* sites. The slots (of the multiple access communication channel) are of unit length. We leave the unit of timing unspecified; all computation times and communication costs are in the same time unit.

The metric chosen to compare the performance of different algorithms or different parameter settings is the *Success Ratio*. 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.

We also evaluated competing strategies with respect to their overheads. The overhead of the algorithm is measured 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. Where necessary, we show the average cost incurred by a successful task set as well as for the failure cases.

Here is a list of the experiments discussed here.

1. Effect of different strategies for determining communication graphs. The performance of the scheme discussed in Section 3.3.1 is evaluated and is also compared against a random scheme where the decision concerning whether or not to allocate a pair of subtasks to the same site is made by the toss of a coin.

2. Effect of incremental changes to CF values. Recall that the algorithm works iteratively by modifying the value of CF if the communication graph generated with the previous value of CF does not produce a feasible schedule. Clearly, the magnitude of the incremental changes made to CF can affect performance and hence we test the performance of different values of this increment.

3. Effect of deadline-driven search. Here we show that it is important to consider task deadlines during scheduling as opposed to finding a minimal length schedule and then *a posteriori* verifying that the schedule meets deadline requirements.

4. Effect of different ways of ordering ready subtasks at a given point during the search for a feasible schedule. Two techniques are compared: ordering according to increasing latest start times and ordering according to decreasing criticalness. *Criticalness* of a subtask is defined as the length of the longest path from the subtask to the terminal node of the communication graph. (In contrast, the latest start time of a subtask is defined to be the (deadline of the periodic task instance that the subtask belongs to *minus* the length of the longest path from the subtask to the last subtask of the periodic task instance).

5. Effect of backtracking during search. These tests show that if the initial path does not lead to a feasible schedule, it is highly likely that the task set is infeasible. This is because, in this case, even a very large number of backtracks results only in a very small improvement in performance.

We also studied the effect of different settings of the laxity factor, communication ratio, and the redundancy parameters on performance. There were no surprises here and hence we do not show the results. As we would expect, as the communication ratio increases, performance drops. The drop is especially pronounced at lower values of the laxity factor. This is to be expected given that with higher communication costs and lower laxities, the difficulty of finding a feasible schedule increases. With increase in the redundancy ratio and the redundancy number, the overheads of the algorithm increases since more subtasks are being allocated and scheduled. The effect on performance is not very significant since, given the formula for P given in (8), task periods increase as the value of these redundancy parameters increase.

4.1 Effect of different strategies for determining communication graphs

We test the performance of the technique used for allocating a pair of communicating subtasks to the same site based on the communication costs, the computation times of the subtasks, and the communication factor (CF). The performance of this scheme is also compared against a random scheme where the decision concerning whether or not to allocate a pair of subtasks to the same site is made by the toss of a coin.

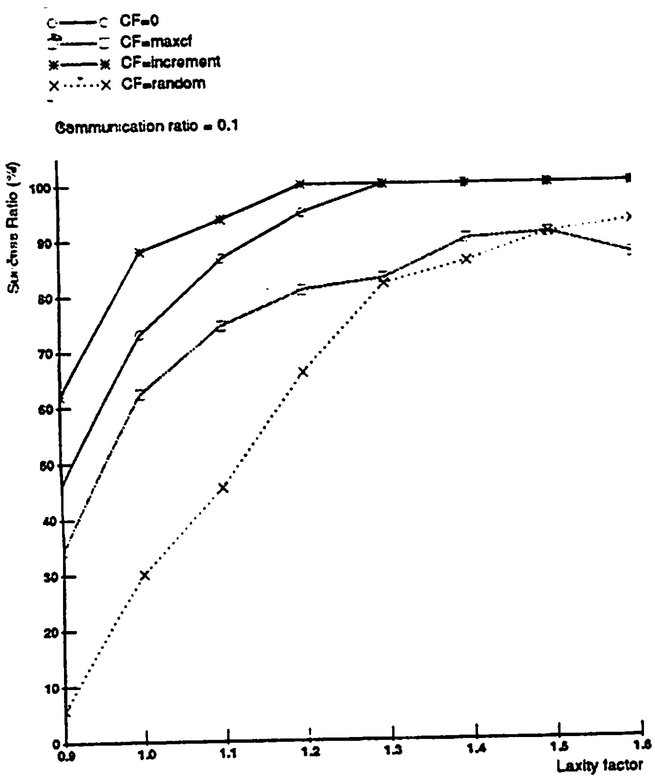
Plots 1.1 and 1.2 show the results for communication ratio values of 0.1 and 0.4 respectively. Each plot has four graphs. The CF=0 graph shows the performance when a pair of communicating subtasks are assigned to *different* sites irrespective of the communication costs or computation times. The CF=maxcf graph shows the performance when a pair of communicating subtasks is assigned to the *same* site irrespective of the communication costs or computation times. (Here all subtasks of a task, except redundant subtasks, are assigned to the same site.) The CF=increment graph shows the performance when communication graphs are generated for different values of CF starting with maxcf down to 0 in increments of maxcf/10. If a feasible schedule is generated for any of these communication graphs, the algorithm is considered to have succeeded with the given task set. The CF=random graph shows the result when communication graphs are generated via coin tosses, one for each pair of communicating subtasks.

The results shown are for the case where the algorithm is *not* allowed to backtrack upon failure.

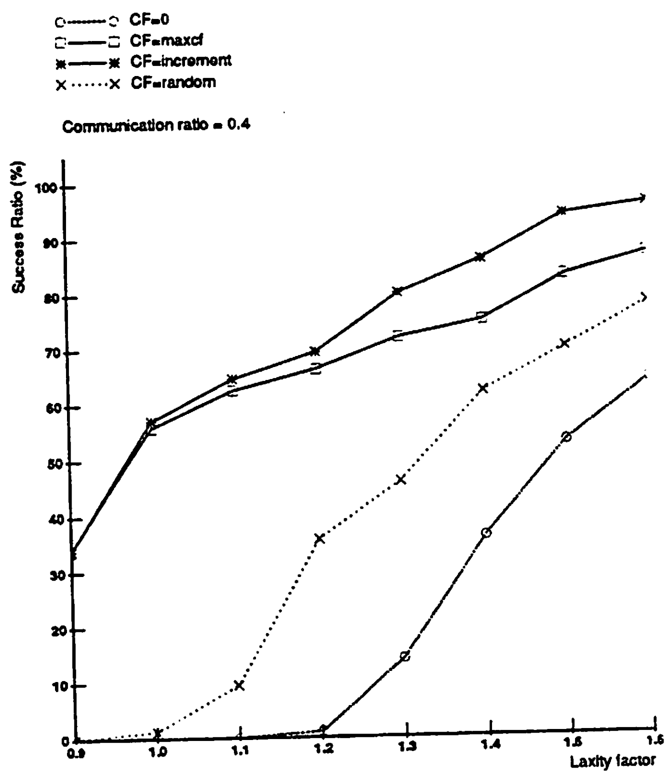
The results show that when communication ratio = 0.4, CF=maxcf performs better than CF=0. The opposite is true when communication ratio = 0.1. The reason should be clear. When communication costs are high (low), it is better to place communicating subtasks on the same site (on different sites). CF=increment performs better than CF=0 and CF=maxcf for both values of communication ratio. This is to be expected since in this case the performance is improved by considering a number of CF values. The poor results with CF=random shows that it is important to make "informed decisions" that make use of subtask characteristics. This is obvious since the CF=random case generally performs worse than the others for communication ratio = 0.1 and worse than CF=maxcf and CF=increment when the ratio is 0.4.

The two plots also show the previously mentioned drop in performance with increasing values of communication ratio and the improvement in performance with increasing task laxities.

Clearly, these results show that it is better to try different communication graphs for different values of CF, especially when communication costs are high or when task laxity is small. However, the more values of CF considered, the more the scheduling costs. Hence we examine this issue in detail next.



Plot 1.1: Strategies for Determining Communication Graphs



Plot 1.2: Strategies for Determining Communication Graphs

4.2 Effect of Incremental Changes to CF Values

Here we show the performance when different increments to CF are considered. Specifically, we study the relative performance when increments are maxcf/5, maxcf/10, and maxcf/20. While some improvement should be expected as we lower the increment, the magnitude of improvement is of interest. Also, the costs of the marginal improvement are of considerable importance.

We should note that when we determine the new communication graph after incrementing the value of CF, we try to determine its schedulability only if (1) the algorithm has not succeeded thus far with the given task set, and (2) the communication graph is different from the most recently considered communication graph. We measure two costs, one for the task sets for which feasible schedules were found, and the other for the rest.

We show the success ratios in plot 2.1 for communication ratios = 0.1 and 0.4. The costs incurred for the test cases that succeeded (failed) are shown in plots 2.2 (2.3).

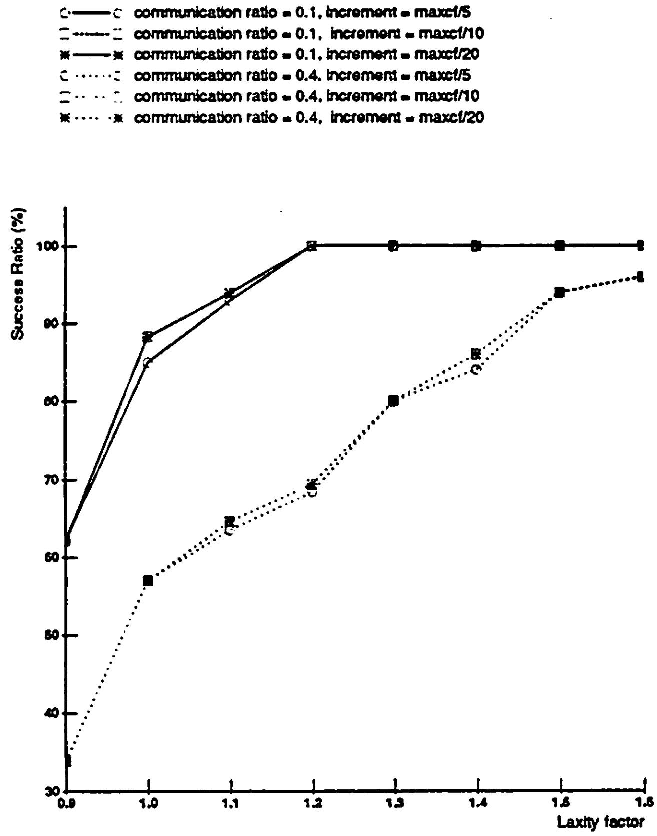
Plot 2.1 shows that, for both values of the communication ratio, there is a very small (less than 3%) performance increase for many values of laxity factor when we go from increments of maxcf/5 to maxcf/10. There is no noticeable further improvement when we go to increments of maxcf/20. The costs show an increase in both cases, even when there is no improvement in performance.

Some explanation of the cost characteristics is in order. When we go from increment=maxcf/10 to increment=maxcf/20 the increase in costs is not proportional to this decrease because even though more communication graphs are generated, a new communication graph is considered only if it is different from the previous communication graph. It appears that on average, the same number of graphs are considered for both values of the increment.

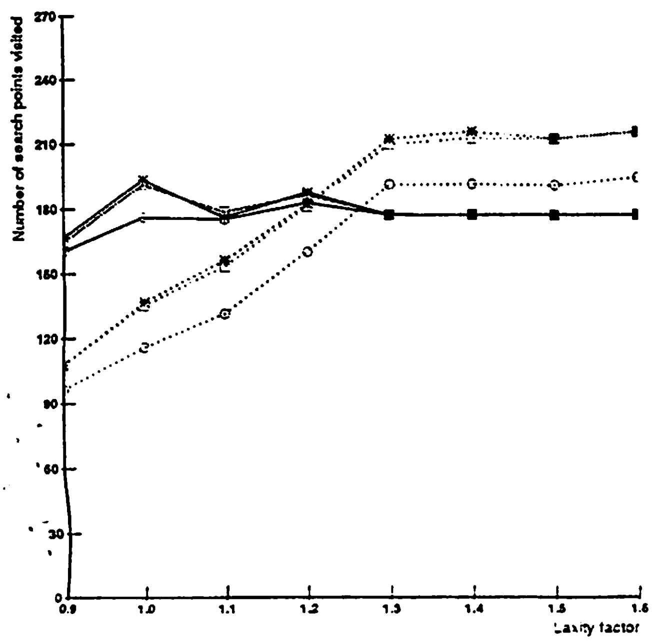
For communication ratio = 0.1, the costs of the success cases cluster in the range 160 to 195. For the failure cases, the cost increases and then falls to 0 at laxity factor 1.2. The latter happens because all task sets are successfully scheduled at laxity factor 1.2. The former is explained shortly. For communication ratio 0.4, the costs continue to increase with the laxity factor except for very large values.

The reason for the increase in costs with the laxity factor is the following: At lower laxity factors, for the failure cases, the algorithm ends with a failure very early in the search. At larger laxity factors, the point of failure occurs after a larger part of the search path has been constructed. Hence the cost increases with the laxity factor for the failure cases. For the success cases, suppose a feasible schedule for a task set is found with the j^{th} communication graph constructed for a task set. Then the cost is the total cost of the failed attempts with communication graphs 1 through j-1 plus the cost of the successful attempt with graph j. Since the former costs increase with the laxity factor, the overall costs of the success cases also increase with the laxity factor.

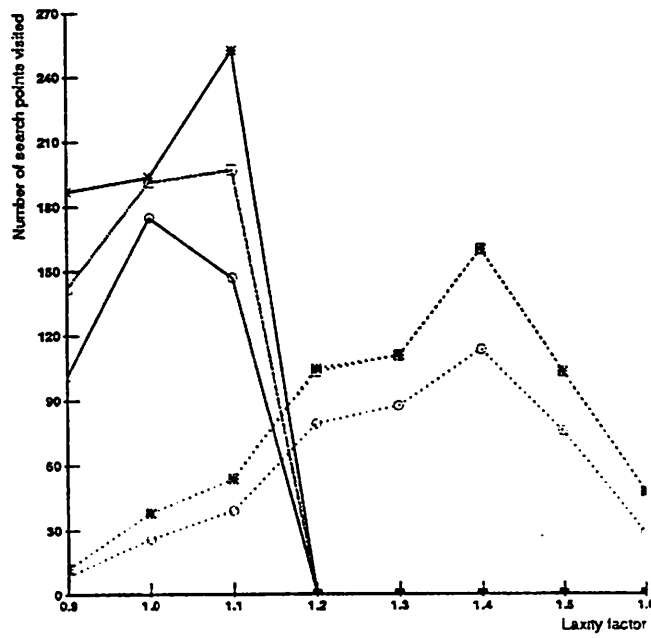
Because of the very small marginal increase in performance with increments=maxcf/20 we work with increments of maxcf/10 for subsequent studies.



Plot 2.1: Effect of Incremental Changes to CF



Plot 2.2: Cost of Incremental Changes to CF - Success Cases



Plot 2.3: Cost of Incremental Changes to CF - Failure Cases

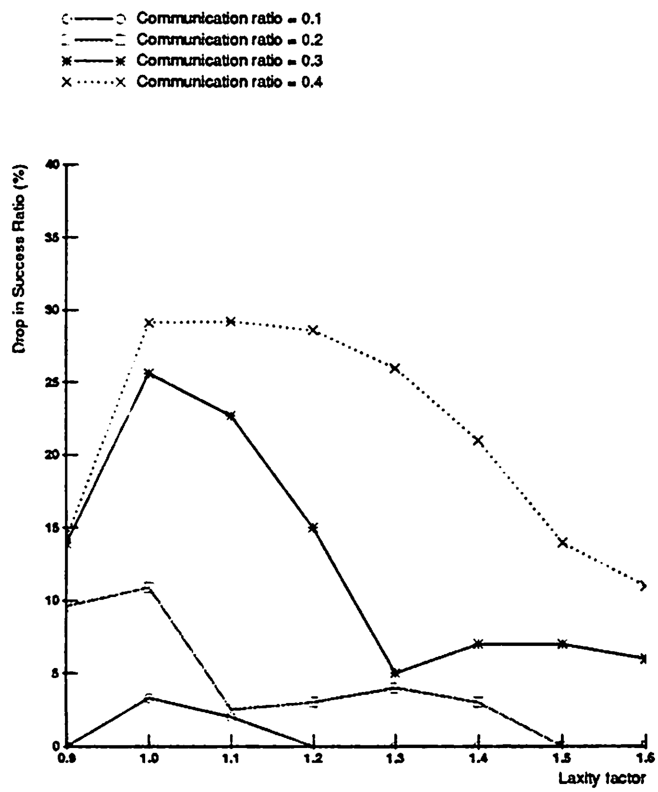
4.3 Effect of Deadline-Driven Search

Suppose we took an approach to allocation and scheduling where deadlines were not considered by the algorithm. Specifically, suppose we use an algorithm, such as in [11] or [6], to construct a schedule that meets communication, precedence, and fault tolerance constraints and finally test the schedule with respect to the deadlines. To simulate such an algorithm, we removed, from our algorithm, the deadline-related conditions used in the check for a valid mapping, and in the check for feasibility. Plot 3.1 shows the drop in the success ratio when these checks are removed. Four graphs are shown, for communication ratio values of 0.1, 0.2, 0.3, and 0.4. The significance of the drops, some as high as 30%, under higher values of communication ratio, indicates that it is important to consider task deadlines during the search as opposed to finding a schedule that meets other constraints and then testing it with respect to task deadlines.

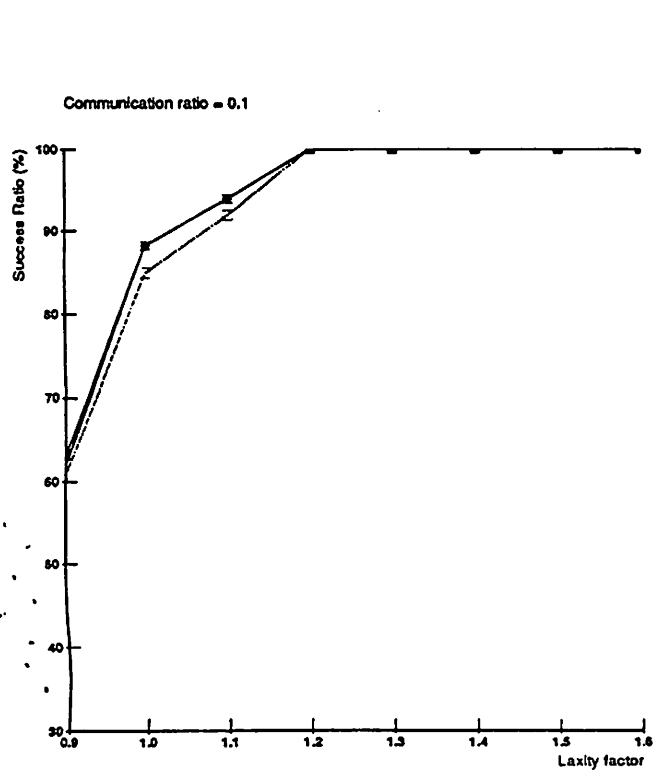
4.4 Effect of Different Ways of Ordering Ready Subtasks

In the tests done thus far, at every point during the search, the ready list is ordered according to the latest start times of the ready subtasks. This was done under the belief that, in general, a subtask which has an earlier latest start time should be considered for scheduling sooner than another that has a later latest start time. Recall that the ordering of the ready list is a crucial component of our mapping algorithm since mappings are generated in an algorithmic fashion according to this order. A different order may imply that a mapping that leads to a feasible schedule will now be generated later in the sequence of mappings. This turned out to be important especially when no backtrackings were allowed.

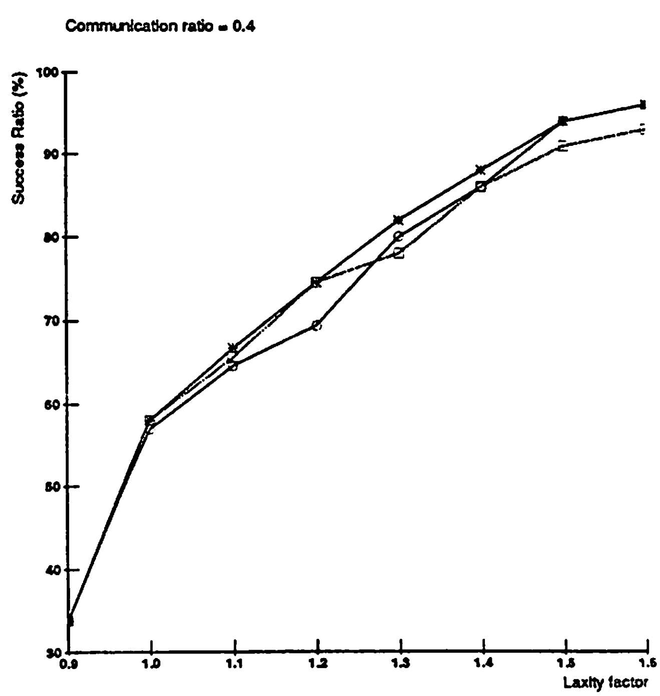
Here we compare the latest start time based ordering with another based on the criticalness of the subtasks. As mentioned earlier, criticalness of a subtasks is the length of the longest path from a subtask to the terminal node of the communication graph. Because of the factors mentioned above, the two orderings displayed slightly different performance characteristics. Plots 4.1 and 4.2 show the performance for two different values of the communication factor. Each plot shows the success ratios for the two orderings as well as the success ratio when one or the other ordering succeeded with a task set (indicated by "deadline + criticalness" in the graphs). Plot 4.1 shows that when communication costs are low, criticalness based ordering has slightly lower performance (around 3%) at low laxity values. When communication costs increase, neither order has superior performance across all laxities, and furthermore, as the "deadline + criticalness" graph indicates, there are a few task sets where scheduling under one ordering succeeds, while the other ordering fails, and vice versa. A closer examination of the situation showed that this was due to minor variations in the two orderings and the aforementioned effect of this on the search without backtracks. The magnitude of the differences drop when backtracking is allowed.



Plot 3.1: Drop in Performance if Deadlines are not used during search



Plot 4.1: Effectiveness of different orderings of ready queue



Plot 4.2: Effectiveness of different orderings of ready queue

4.5 Effect of Backtracking during Search

Whereas the results shown thus far were obtained when the algorithm is not allowed to backtrack, we examine the effect of allowing a limited number of backtracks. Plots 5.1 and 5.4 show the results for two different values of communication ratio. The graphs show that, for the task sets considered, whereas there is only a small improvement of less that 3.5% even at low laxities (i.e., tight situations) when we allow 100 backtracks, allowing even more backtracks does not have a noticeable improvement in performance.

In plots 5.2, 5.3, 5.5, and 5.6, we show the effect of allowing backtracks on the scheduling costs for the successfully scheduled tasks sets as well as for the rest of the task sets. These show that a very large price (as much as a *four* fold increase in costs for successful cases and a *thirty* fold increase for the failure cases) is being paid for obtaining a small marginal increase in performance.

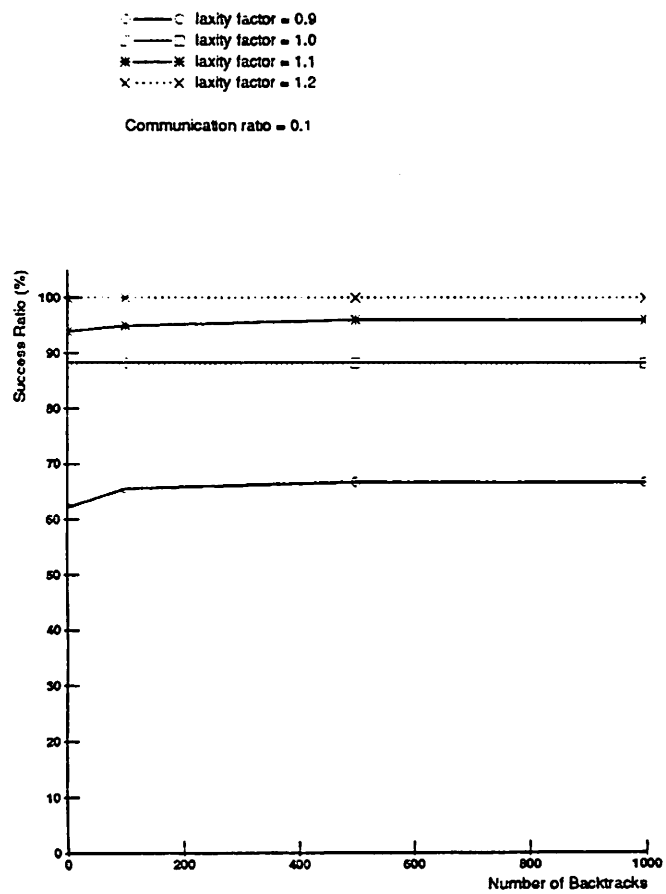
Overall, the tests indicate that if the initial path does not lead to a feasible schedule, it is highly likely that the task set is infeasible. This is because, a very large number of backtracks results only in a very small improvement in performance and at a very high marginal increase in cost. These results show the effectiveness of the schemes used in determining the initial search path.

5 Extensions to the Algorithm

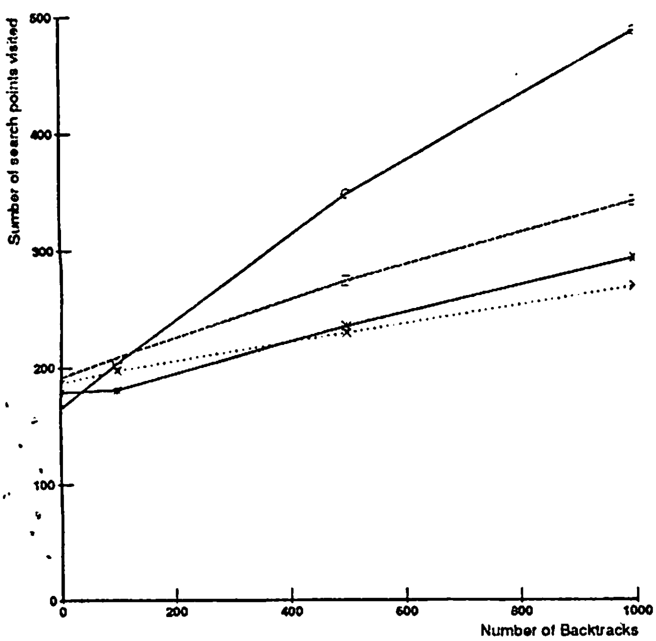
Whereas the specific algorithm discussed so far handles the constraints mentioned in Section 2, the general framework is capable of dealing with additional constraints, more complicated resources, including communication resources, and more sophisticated search schemes. In this section we discuss some of the issues that we are continuing to investigate by enhancing the implemented algorithm.

5.1 Parallelism within a Site

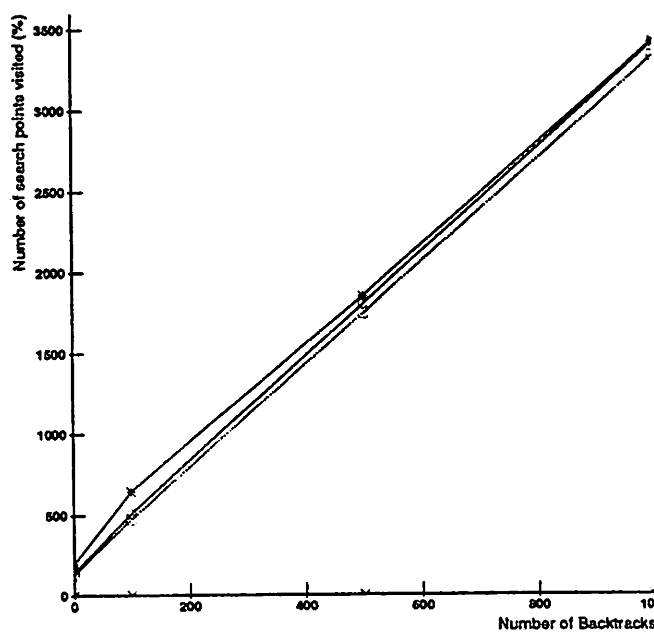
The resource model discussed until this point is one where each site has one processor as well as a number of (passive) resources, such as files. It was assumed that all the resources on a site are dedicated to the subtask executing at any given point. In other words, there is no parallelism within a site. Because of this, allocation of a subtask to a site and its scheduling on that site was handled using a very simple technique: A site is considered idle if no task is scheduled to execute on it at the current time. A subtask can be allocated to a site if the resources needed by the (enabled) subtask are available on that site and the site is idle. If instead, idleness of a site is determined with respect to the resources needed by a task, we can improve parallelism in the following way. A site can be considered for scheduling if at least one of its schedulable resources (such as CPU and devices) is idle. A subtask can be scheduled on a site if the site is idle with respect to the resources needed by that subtask.



Plot 5.1: Effect of Backtracking



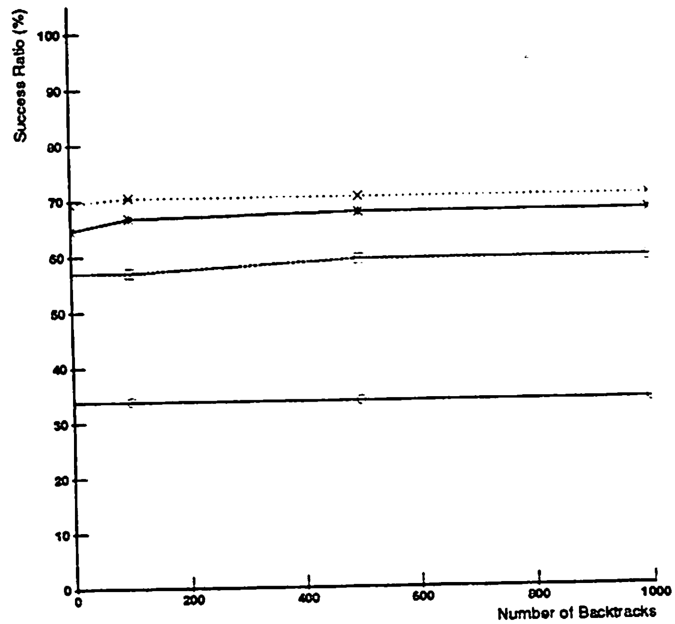
Plot 5.2: Cost of Backtracking -- Success Cases



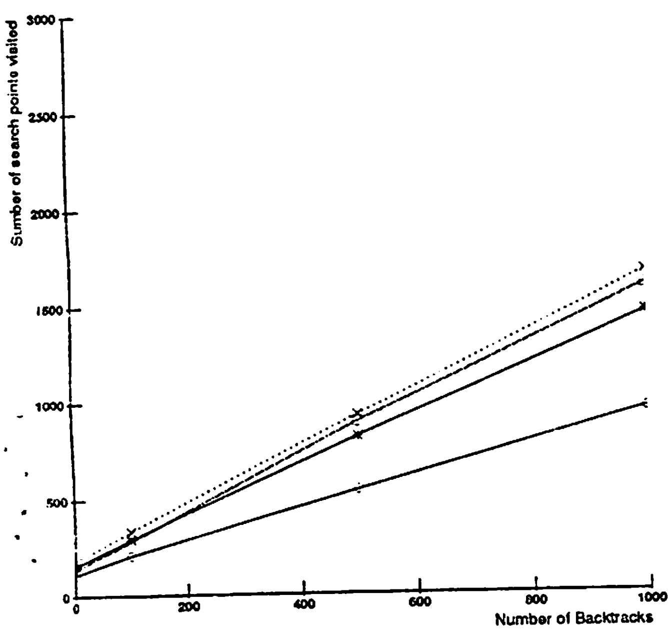
Plot 5.3: Cost of Backtracking -- Failure Cases

○——○ laxity factor = 0.9
□——□ laxity factor = 1.0
✱——✱ laxity factor = 1.1
×······× laxity factor = 1.2

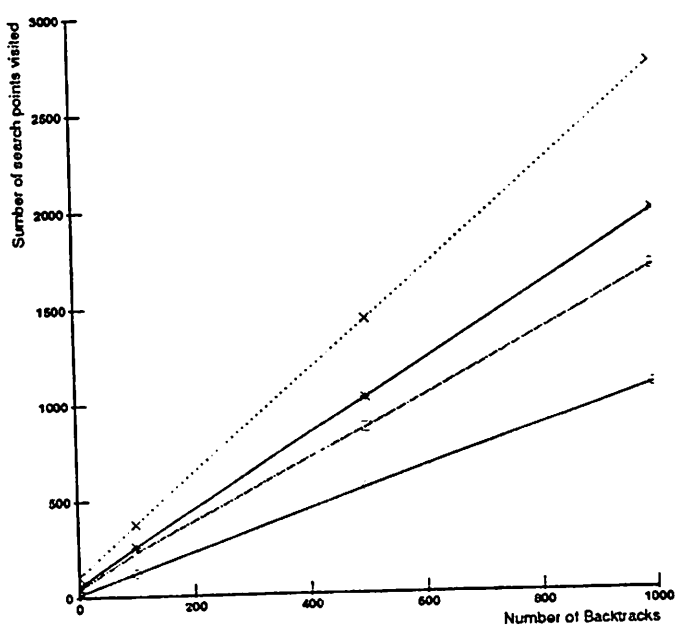
Communication ratio = 0.4



Plot 5.4: Effect of Backtracking



Plot 5.5: Cost of Backtracking – Success Cases



Plot 5.6: Cost of Backtracking – Failure Cases

5.2 Consistency of Replicated Shared Resources

So far, we have not dealt with resource consistency in any detail. When replicated shared resources are involved, tasks should be allowed concurrent access to resources such that the consistency of the resources is not violated. When resource sharing occurs only among instances of a task, the ordering of instances will implicitly ensure the consistency of the resources. But when resource sharing occurs across tasks and some of the subtasks are replicated (for fault tolerance), consistency-preserving ordering must be explicitly ensured. This will take the form of *additional* conditions imposed on the validity of a mapping: Replicates of subtasks should access replicated shared (writable) resources in the same order.

5.3 Communication Topologies/Protocols

The effect of communication schemes that are more complicated than the one assumed so far is discussed below:

- The scheme discussed thus far can also be applied when sites are connected by a point-to-point communication network. In this case, however, an additional condition will be imposed on the validity of a mapping: A pair of communicating subtasks should be allocated to sites connected by a communication path.

- In TDMA (Time Division Multiple Access) networks, each site gets access rights (say, for w units of time) once during every cycle (say, of length c). Hence with TDMA, the minimum worst-case delay is at least c. In general, the worst-case delay is some multiple of c depending on whether the information from a subtask to its successor is transferable in time w. In any case, such delays are computable and used in the construction of *communication graphs*. The scheduling of the (multiple access) network is also not too much more complicated than the simple scheme: A TDMA gives the impression of each site possessing an exclusive communication channel (where each can be used between specific intervals only - depending on the phasing of the sites).

- In deterministic protocols using dynamic token passing schemes [15] [9] the maximum communication delays can be determined and can be used in the the construction of *communication graphs*. However, in this case, the algorithm will not be involved in scheduling communication subtasks.

The model of real-time systems assumed in this paper is one in which subtasks of periodic tasks are preallocated to sites and their start times are predetermined. Because of this, probabilistic protocols such as the Ethernet are not appropriate for consideration in this context since they do not guarantee message delivery within bounded intervals.

5.4 Using Dynamic Task Characteristics during Search

Recall that at each point in the search, the latest start time of tasks is utilized for determining the order in which ready tasks are scheduled. The advantage of this scheme is that task characteristics do not have to be reevaluated as search progresses. However, it is likely that use of dynamic task features will offer benefits.

A specific idea that appears worth pursuing is the following: Construct a heuristic function along the lines of [16] that is flexible enough to account for different (sub)task characteristics, such as deadlines, computation times, communication delays and resource requirements. Rather than using the *least latest start time first* approach use a *least heuristic value first* approach for ordering the ready list. That is, the heuristic function is applied to each of the tasks that is ready and the ready list is organized in an increasing order of these values.

5.5 Scheduling Aperiodic Tasks

So far we have focussed on the scheduling of periodic tasks. In most real-time systems, aperiodic tasks do occur even though they may not be as complex as the periodic tasks we have considered up to now. In this section we discuss simple aperiodic tasks, in particular, tasks that execute on a single site.

One of the characteristics of the current scheduling algorithm is that it produces a "packed" rigid schedule on each site. This is because a subtask is scheduled at the earliest time possible. While this results in early finish times for subtasks, it causes idle times to occur as late as possible and to be clustered together. This may reduce the chances of scheduling (dynamically arriving) aperiodic tasks. It would be better if the idle times were more evenly distributed. However, an even redistribution of idle times will affect all sites because of the ripple effect of one site's schedule on other sites. Here we propose a scheme that is local in its effect. It is subject to the following constraints.

- The schedule for any subtask that just meets its deadline (i.e., its completion time as per the schedule is equal to its deadline) can not be postponed.
- The schedule for a subtask that has a successor on another site can not be moved forward in time(otherwise it may affect the schedule of its successor on a different site).
- The schedule for a subtask that has a predecessor on another site can not be moved forward (for similar reasons as above).

The scheme works as follows: Given the schedule for a site, *maximal subschedules* are determined such that the schedule for the first subtask in the subschedule can not be moved forward and that of the last subtask can not be moved back. Ls, the length of the subschedule is given by the length of the interval from the earliest start time of the first subtask to the latest finish time of the last subtask. Cs is the sum of the computation times of subtasks in the subschedule. $(Ls - Cs)$ is the amount of free time within the subschedule. The algorithm can proceed in one of two different ways:

1. Assign *scheduling windows* to each subtask (in the subschedule) of length $\frac{C \times Ls}{Cs}$, where C is the computation time of the subtask. Aperiodic tasks can be scheduled as long as the subtasks are scheduled to execute within their assigned scheduling windows.

2. When aperiodic tasks are scheduled ensure that the set of subtasks in a subschedule are executed within the interval of length Ls subject to the precedence constraints of these subtasks.

The second scheme provides more leeway for aperiodic task scheduling but demands consideration of precedence constraints while scheduling these tasks.

The above technique attempts to schedule aperiodic tasks when they arrive. Another possibility, within the framework of periodic tasks is to allocate a periodic task to each site with the specific purpose of "reserving" resources for dynamically arriving aperiodic tasks. Such tasks will be executed within the intervals allocated for the periodic task. In this case, various improvements, as suggested in [8] can be applied to improve the schedulability of aperiodic tasks.

6 Conclusion

The algorithm discussed in this paper for scheduling periodic tasks provides a framework for allocating and scheduling complex real-time tasks in distributed and multiprocessor systems. It has the following features:

- It is based on a search technique that takes into account the various task characteristics. Its decisions relate to the site assignment of each subtask of a task, the order in which each site processes its subtasks, and how the communication medium schedules message transmissions.

- It allocates and schedules subtasks whose execution is constrained by precedence constraints, communication requirements, replication needs arising from fault tolerance requirements, resource requirements, and consistency constraints.

- Periodicity constraints of tasks are translated into deadline constraints of subtasks of these periodic tasks. Determining a schedule that meets these deadlines while satisfying the other constraints is the goal of the algorithm. This is a fundamental difference between this (deadline-driven) algorithm and other similar ones (e.g. [6], [11]) that deal with precedence constrained scheduling aimed at producing least length schedules.

- The choice of the next path to follow and whether or not to continue along a path are determined by checking whether the (partial) schedule represented by the path is likely to lead to a feasible schedule (one that meets timing, resource, placement, and precedence specifications of the subtasks).

- It facilitates the coexistence of fault masking and forward error recovery techniques by allowing the replication of some of the subtasks.

- It permits communication between subtasks to occur via statically scheduled communication links as well as using protocols with predictable maximum communication delays.

- It is general enough to utilize a spectrum of branching rules, from the simple rules based on the static properties of subtasks, for example, the latest start time of tasks, number of successors, etc., to complex branching rules based on the dynamic properties of tasks, in particular, the requirements of the subtasks that are yet to be scheduled.

- It is designed to determine a feasible schedule without backtracking. But it allows for backtracking. As the experimental results show, in a vast majority of situations, if a feasible allocation and schedule exists, it can be found without any backtracking.

In this paper we presented a scheme for generating the communication graph that was based on a pair-wise examination of communicating subtasks. Currently we are developing and evaluating other schemes for the clustering of subtasks.

The algorithm presented here is particularly suitable for determining the allocation and scheduling of periodic tasks at preallocation time [19]. When a system goes through different modes of operation, it is the case that not all periodic tasks occur in all the modes. One straight-forward way to use the algorithm in this paper is to allocate and schedule *all* the periodic tasks at system initialization time. Of course, this will produce an underutilized system. A better static scheme is to design a set of schedules, each applicable to a specific mode; schedules are switched when a mode change occurs. In this case, further constraints will be imposed on the schedules constructed upon mode changes. These will typically be placement constraints arising from the need to maintain continuity (with respect to the schedule adopted in the previous mode) of allocation of a subtask to a specific site.

As discussed in Section 5.5, even though the algorithm is primarily concerned with periodic tasks, it can be extended to schedule aperiodic tasks that execute on a single site. Adjustment of the schedule produced by the algorithm to enhance the schedulability of such tasks was also discussed. With this enhancement, the present algorithm for scheduling periodic tasks can be used in conjunction with a decentralized algorithm for allocating and scheduling aperiodic tasks [14].

Acknowledgements

Comments on previous versions of this paper by Christian Koza and Dilip Kandlur as well as the following members of the Spring Project are gratefully acknowledged: John Stankovic, Chia Shen, FuXing Wang, Doug Niehaus, Lory Molesky, Arvind, and Goran Zlokapa.

References

[1] J. A. Bannister and K. S. Trivedi. Task allocation in fault-tolerant distributed systems. In *Acta Informatica*, 20, Springer-Verlag, 1983.

[2] S. Cheng, 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.

[3] E.B. Fernandez, and Bussell, B., Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules, *Communications of the ACM*, C-22, Aug 1973, 745-751.

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

[5] T.C.Hu. Parallel Sequencing and Assembly Line Problem, *Operations Research*, vol 9, 841-848, Nov 1961.

[6] H. Kasahara and S. Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing, *IEEE Transactions on Computers*, vol C-33, 1,, November 1984, 1023-1029.

[7] H. Kasahara and S. Narita. Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System, *IEEE Journal of Robotics and Automation*, RA-1(2), June 1985.

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

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

[10] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20(1), 1973.

[11] P.-Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computer*, C–31(1), 1982.

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

[13] D.T.Peng 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.

[14] K. Ramamritham, 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.

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

[16] K. Ramamritham, J. Stankovic, P. Shiah, "O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems," *International Conference on Parallel Processing Systems*, August 1989.

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

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

[19] J.A. Stankovic, and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real–Time Operating Systems," *ACM Operating Systems Review*, Vol. 23, No. 3. July, 1989, pp. 54-71.

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

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

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