SCHEDULING GROUPS OF TASKS IN DISTRIBUTED HARD REAL-TIME SYSTEMS

Sheng-Chang Cheng
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

John A. Stankovic and Krithivasan Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 87-121
November 9, 1987

Scheduling Groups of Tasks in Distributed Hard Real-Time Systems *

Sheng-Chang Cheng †
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

John A. Stankovic
Krithivasan Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

November 9, 1987

*This work was supported, in part, by the Office of Naval Research under Grant 048-716/3-2285 and by National Science Foundation under Grant DCR-8500332.

†This work was performed as part of the Spring project at the University of Massachusetts. The first author is now with IBM, Yorktown Heights.

1 Introduction

In the next generation hard real-time systems, it is anticipated that many applications will contain collections of tasks running on a distributed system, and that many of these tasks must be executed according to certain precedence and real–time constraints. The Space station, new process control applications integrated with expert systems software, other real–time AI applications such as an autonomous land rover, and SDI are some examples of such future applications. In these next generation, large-scale, hard real-time systems, one will find tasks with differing functional complexity and multiple levels of timing constraints. For example, processing data from a sensor is sometimes functionally simple, and might need to occur in the microsecond range, while consistent updates of replicated files is more complex and might need to occur in the seconds range. At even higher levels, control and coordination tasks might be quite intricate and need to occur in the seconds or minutes range, and movement of material in an automated factory might be based on complicated coordination algorithms, but have deadlines in the range of minutes or even hours. The work presented in this paper is applicable to the groups of tasks where their deadlines are in the one half second range or greater, and, thus, does not address tight timing constraints such as those that occur in sensor processing. We assume that the system under consideration has lower level schedulers and mechanisms (such as dedicated processors) to handle tasks with very tight time constraints. Hence, a complex system will have multiple levels of scheduling based on the granularity of the timing constraint. This paper deals only with collections of high level tasks that have precedence constraints and relatively long deadlines.

Scheduling such task groups is more complicated than scheduling individual, independent tasks. To determine an optimal schedule for a group of tasks with arbitrary precedence constraints on multiple processors has been shown to be *NP-hard* [Ull76]. In this paper, we develop heuristics for dynamically scheduling such groups in a distributed hard real-time system. Each group has a single deadline and tasks in a given group have user-specified precedence constraints among themselves. We assume that the task groups are invoked dynamically and that the system is too complicated to statically precompute schedules for all possible combinations of such task groups.

The primary contributions of this paper are:

- An atomic guarantee algorithm that can handle arbitrary types of precedence constraints among tasks: This algorithm is based on a partition scheme. To reduce scheduling delays, the algorithm divides tasks in a group into subgroups and distributes the subgroups across nodes to be scheduled. This scheme allows the system to schedule tasks in a group in parallel and increases the chance of scheduling a distributed group.

- Performance evaluation of the algorithm and comparison with two alternative algorithms via simulation: The simulation results show that:

  - The algorithm is very effective for a wide range of system conditions, and for groups of different types of precedence constraints and different sizes.
  - The algorithm performs better than alternative algorithms in the majority of the system conditions and for different types of groups.

- Dynamic and distributed scheduling of task groups is practical for applications with a wide range of characteristics.
- A distributed hard real-time system can benefit substantially from distributing tasks in groups across the network.

The remainder of the paper is organized as follows: Section 2 provides background information to point out that the problem addressed in this paper has not been adequately dealt with before. Section 3 presents a model of the system under consideration, the nature of task groups in this system, and describes the scheduling problem in depth. Section 4 discusses two simple atomic guarantee algorithms to serve as a basis for performance comparison and to motivate a more sophisticated algorithm called the partition atomic guarantee algorithm. Section 5 describes the partition atomic guarantee algorithm in detail. Section 6 describes the evaluation approach used to analyze the guarantee algorithms. It presents the simulation model, the baseline algorithms, and the performance metrics. The simulation results themselves are reported in Section 7. A summary is given in Section 8.

2 Background

A large body of research results exist in the area of scheduling in hard real–time systems. For a survey see [CSR87]. Here we examine previous work on scheduling in the presence of precedence constraints.

Most research on scheduling nonpreemptive tasks with precedence constraints is restricted to special cases and is limited to centralized systems. For example, Lawler [Law73] developed an $O(n^2)$ algorithm to schedule tasks on uniprocessor systems. Garey and Johnson [GJ76] developed an $O(n^2)$ algorithm to schedule tasks with unit computation times on two-processor systems. Hu [Hu61] developed an $O(n \log n)$ algorithm to schedule tasks with unit computation times and with tree-like precedence constraints on multiprocessor systems. For more general cases, the scheduling problem becomes *NP-hard* and hence computationally intractable.

Many researchers have attempted to solve the general case by using an exhaustive search approach. For example, Ramamoorthy *et. al.* [RCG72] used a dynamic programming method to find the optimal schedule for tasks with arbitrary precedence constraints on multiprocessor systems. Also, Ma *et. al.* [MLT82] developed an integer programming model to allocate time-critical application tasks on multiprocessor systems. More recently, Kasahara and Marita [KN85] have used a heuristic driven depth first algorithm for static scheduling of tasks with arbitrary precedence constraints in a multiprocessor. Because of the exponential computational cost in the worst case, these approaches cannot be used for dynamic scheduling.

Leinbaugh and Yamini [LY82] derived the worst case finish time for a set of tasks which run on a dedicated network. Each task consists of multiple segments. By assuming that each task is blocked by every other possible task due to both resource sharing and inter-task communication, they compute the maximum blocking time for each task. Their approach requires complete knowledge of tasks present in the entire system. Their results are intended to serve as a upper bound on task response time for static systems.

Dynamic scheduling of task groups in distributed systems is much more complicated than that in a centralized system or a static distributed system. Also, due to communication delays, nodes must make scheduling decisions based on out-of-date state information about other nodes. Because of these complications, scheduling algorithms for such systems often consist of two components: a local scheduling component for dynamically deciding whether tasks arriving at a node can be scheduled on that node and a distributed scheduling algorithm for dynamically deciding where, in the network, tasks not scheduled locally should be assigned. For distributed scheduling, Ramamritham and Stankovic [RS84] developed an algorithm for scheduling independent tasks in distributed hard real-time systems. Their approach combines focussed addressing and bidding. Here, we extend this algorithm to handle distributed scheduling of task groups.

3 System and Task Models

In this section, we describe the group scheduling problem in detail. We describe the model of a distributed real-time system and the nature of task groups.

3.1 System Model

A system consists of a number of geographically distributed nodes interconnected by a communication network. The nodes dynamically interact with the external environment. Each node contains a processor for executing application tasks and a co-processor for executing system scheduling tasks. The co-processor is utilized to off-load scheduling overhead and to simplify the scheduling process. We expect each node to have considerable multiprocessing power and hence believe that in future hard real-time systems the cost of the co-processor will be negligible. This approach offers huge dividends because it will enable the use of on-line scheduling algorithms which have the potential to improve flexibility and lower testing costs. Note that spare cycles on the co-processor could also be used to run application tasks; this involves a simple extension to the local scheduling strategy discussed in this paper.

Because scheduling groups of tasks with precedence constraints in a distributed system is a difficult problem by itself, we confine ourselves to the case where each node has only one processor for executing application tasks, i.e., we do not consider the situation where multiple processors are available for executing application tasks on a node. We also assume that nodes in the network are homogeneous in the sense that they have the same capability to execute tasks, so that a task can be executed on any node in the system.

3.2 Nature of Task Groups

Task groups are nonperiodic and may exist at any node in the network. Tasks in each group are interrelated by arbitrary precedence constraints and the precedence graph of a task group is assumed to be an acyclic directed graph. All tasks in a group have to be executed or no tasks should be started. Each group must finish by a specified deadline. Tasks are nonpreemptable. For simplicity, we assume that all resources required by tasks except the CPU are always available.

Each task in a group is specified by its precedence constraints and its worst case computation time. Each task may have multiple predecessors and multiple successors. A group has one or more beginning tasks and ending tasks. A *beginning task* is one with no predecessors and an *ending task* is one with no successors. The node where a task group originally arrives is called the *originating node*. This node decides whether the group can be guaranteed or not. If a group is guaranteed, the originating node sends a message to each node where a beginning task is scheduled to activate the group. The time required for guaranteeing a group followed by its activation is accounted for by the scheduling algorithm. [1]

Once a group begins execution, tasks in the group must communicate with each other in real-time. When a predecessor task finishes, it sends an *enabling message*, as well as output data, to each of its successors. A successor is *enabled*, i.e., can begin execution, only after the enabling messages from its predecessors have been received. In hard real-time systems, the time spent in communication between tasks must be accounted for explicitly in scheduling a group. We assume that the communication network is designed in such a way that the communication time for messages exchanged between tasks executed on different nodes are bounded. The communication time between tasks executed on the same node is assumed to be zero.

As an example, a sample task group is shown in Figure 1. The group contains five tasks and is specified by a precedence graph. In this group, the computation time of different tasks is the same, which is 10 time units. The worst case communication time between tasks executed at different nodes are marked on the edge of each pair of communicating tasks. This example task group will be used throughout the paper. For purpose of the running example, the deadline of the sample group is assumed to be 300.

3.3 Main Features Of Approach

The main features of the distributed scheduling approach described in this paper are:

- When a new task group is invoked, the system immediately attempts to determine whether it can guarantee that all tasks in the group will execute within the specified timing constraints. This feature gives the system more leeway in applying compensating actions if the task group cannot be guaranteed.
- Either all tasks in the group are guaranteed to execute or none execute. This feature is termed *atomic guarantee*.
- Timing constraints are explicitly addressed. This is in contrast with the standard approach of translating timing constraints into task priorities and using priority-based scheduling.
- The scheduling algorithm itself operates in parallel to reduce the delay in scheduling, thereby increasing the probability of guaranteeing a group. Scheduling and communication overheads are taken into consideration.

[1] This can be accomplished in the following way: Each group is guaranteed with a dummy task scheduled on the originating node to run before the beginning tasks. The dummy task sends enabling messages to activate the beginning tasks. The communication time between the dummy task and the beginning tasks is included in the precedence graph of the group.

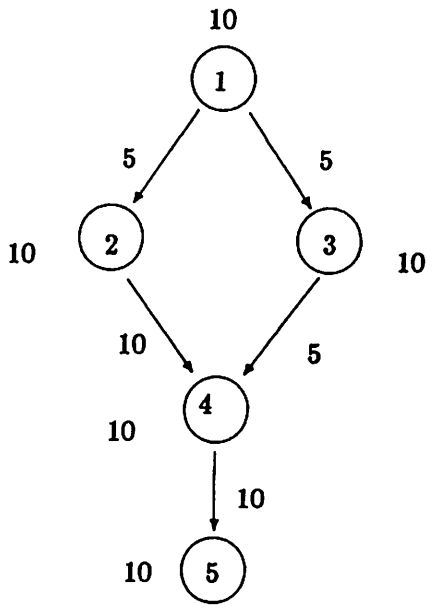


Figure 1: A sample task group.

- Certain subsets of tasks in a group are clustered, i.e., forced to be scheduled on a single node, as a function of communication requirements, the deadline of the group, and the current state of the network.
- Clusters of tasks are sometimes distributed across nodes in the network thereby utilizing the inherent parallelism of the distributed system. Decisions concerning this distribution are made as a function of the current estimated surplus processing power of nodes in the network.

4 Atomic Guarantee Algorithms

When a task group arrives at a node, the node needs to decide whether tasks in the group can be guaranteed locally. If the entire group can be guaranteed locally, the group is assigned to the node. Otherwise, the node needs to decide how to distribute tasks in the group among nodes in the network such that the group can be guaranteed to complete before its deadline.

In this section, we present two heuristic guarantee algorithms. The first one is called the *sequential* algorithm which attempts to schedule tasks on one node at a time. The second one is called the *polling* algorithm which attempts to poll multiple nodes to schedule tasks. The discussions of these atomic guarantee algorithms is included to serve as a basis for performance comparisons and to reveal the need for developing a more effective algorithm. Such an algorithm is developed in section 5. Problems with the simple algorithms presented in this section are highlighted.

4.1 The Local Scheduling Scheme

Determining the schedule for a set of nonpreemptable tasks on a node is a *NP-hard* problem [Ull76]. Given that we are interested in dynamic scheduling, we use a heuristic approach to solve this problem.

To guarantee a task group on a node, the node selects tasks of the group one at a time in a reverse topological order and invokes a local scheduler routine to compute the latest start time of each task taking into account its precedence constraints as well as the constraints imposed by all other previously guaranteed tasks at this node. If all tasks of the group can be guaranteed then the entire group is guaranteed in an atomic fashion and activated. Otherwise, distributed scheduling is invoked.

Since the local scheduling algorithm is a simple extension to previously published work [RS84,SRC85] and since this paper concentrates on distributed scheduling we do not provide the details on the local scheduler itself. See [SRC85] for details. However, note that in all subsequent discussions of the distributed algorithms, the local scheduler is invoked both initially when a task group arrives and later on the remote node when members of a task group are transmitted to that remote node.

4.2 Simple Distributed Guarantee Algorithms

Since tasks do not have individual deadlines, a potential approach to performing distributed scheduling is for the originating node to perform the following:

1. Sort tasks in the group in reverse topological order. Send information about the first task, i, to selected nodes.

2. Assign a *pseudo deadline* to task i for each selected node as follows:

$$D_i = \min_j(LST_j - Comm(i,j))$$

   where LST_j is the latest start time of a successor j and $Comm(i,j)$ is the communication time between i and j. If tasks i and j are sent to the same node, the communication time is zero. If i is an ending task, D_i equals the deadline of the group.

3. Determine the *latest start time* (LST) of task i on each selected node with respect to the complete set of tasks already scheduled on that node. If the latest start time of the task being scheduled is greater than the current time, the task is *tentatively* scheduled on the node. Reserve processing power on the node for the scheduled task. LST_i is the largest of such LSTs.

4. Repeat steps 1 to 3 until every task in the group is scheduled or an unschedulable task is found. Guarantee the group if every task is scheduled and make the reservation permanent. Otherwise, abort the group and release the reserved processing power.

4.2.1 The Sequential Algorithm

The first atomic guarantee algorithm, called the *sequential algorithm*, is straightforward. To distribute a group of tasks, the originating node first attempts to schedule the tasks locally, one at a time and in a reverse topological order. It attempts to schedule as many tasks as possible until it determines that a task in the task group is not schedulable. At that time, it keeps the tasks which are tentatively scheduled and sends the unscheduled tasks to another node whose surplus processing power is greater than a threshold, NSF (*Node Surplus Fraction*), a system tunable parameter. The same scenario is then repeated at the recipient node: After the recipient node has attempted to schedule the tasks sent to it, any remnant of the group is sent to yet another node until either all the tasks are scheduled or the group has visited all the nodes with surplus greater than NSF. If all the tasks in the group have been scheduled, the originating node guarantees and activates the group. Otherwise, the node aborts the group.

To estimate each other's surplus processing power, each node in the network transmits surplus information to other nodes periodically; each node maintains the information in a *surplus window*, which is an interval between the current time and a future time. The size of the surplus window is a tunable parameter of the system.

In the sequential algorithm, transmission of tasks is sequential. Tasks remaining to be scheduled are sent to one node at a time. No attempt is made to distribute tasks in parallel. The advantage of this algorithm is its simplicity and its low processing overhead. However, if a task is scheduled on a busy node, its LST may be so close to the current time that the time left for its predecessors is small and it therefore becomes difficult to guarantee the group. Another disadvantage is the delay involved in processing a group, since tasks in the group are processed sequentially one at a time and on a node at a time. Hence, we now hypothesize (and later show) that this simple approach will perform poorly.

4.2.2 The Polling Algorithm

The second atomic guarantee algorithm is based on a polling scheme: The originating node sends information about a single task to a number of nodes with surplus processing power greater than NSF, and requests the set of nodes to attempt to schedule the task. Tasks are chosen one at a time and in a reverse topological order. Each node reserves processing power for each scheduled task and returns the information about the task's latest start time to the originating node. For each task, the originating node selects the node that returns the greatest LST and sends the task to the selected node. Other responding nodes are told to release the reserved processing power. If, when a task arrives at a selected node, the latest start time of the task is still greater than the current time, the node sends a *confirmation* back to the originating node. The originating node guarantees a group if every task is scheduled and confirmed.

The advantage of this algorithm is its relative simplicity since one task at a time is considered. Another advantage is that, if a large number of nodes are polled, the node offering the best LST can be determined, thereby increasing the schedulability of the group. However, this algorithm has several weaknesses: (1) Because a polled node needs to reserve processing power for tasks, if a large number of nodes are polled, the attempt to schedule different tasks (either from the same group or different groups) on the same node may have

severe conflicts, (2) The scheduling costs and the communication costs are large, because the attempt to schedule tasks is replicated at multiple nodes, and (3) The scheduling delay is large because tasks are scheduled sequentially and because, for each task, the originating node waits for response from multiple nodes. It is difficult to hypothesize just how well the polling algorithm will perform. The evaluation shows that it sometimes performs well and at the other times performs poorly.

To overcome the weaknesses of the two simple algorithms mentioned so far, we have developed a sophisticated atomic guarantee algorithm, called the *partition algorithm*. This algorithm is described in the next section and we later show its superior performance.

5 The Partition Guarantee Algorithm

The partition guarantee algorithm is invoked when all tasks in a group cannot be guaranteed locally. To reduce scheduling delay and to utilize the inherent parallelism of a distributed system, the algorithm processes tasks in a group in parallel (instead of in a reverse topological order). Since tasks must be executed according to precedence constraints, it is necessary to assign a pseudo time frame to each task wherein that task can be scheduled to run so that a parallel scheduling effort can proceed. This time frame is called the *scheduling window* and it is only an initial approximation. Since we assume that tasks executing on the same node have zero communication costs, the algorithm attempts to cluster highly communicating tasks and then treats the cluster as a single entity in attemps to send parts of a group to other nodes. To determine how to distribute the clusters in the network, a scheme that combines *focussed addressing* and *bidding* is used. Decisions concerning this distribution are made as a function of the current surplus processing power of nodes in the network. If a node involved in scheduling any cluster of the original group determines that it is impossible to schedule it within the initially determined scheduling windows, it communicates with other nodes in order to adjust the scheduling windows since they were only approximations. Once the originating node determines that, for every cluster, there exists a node that guarantees the execution of all the tasks in the cluster within their *last* assigned windows, it guarantees the execution of the group and activates the group.

We now describe the details of the main components of the partition algorithm: assigning scheduling windows, task clustering, distributing clusters and adjusting the windows.

5.1 Assigning Scheduling Windows

The *scheduling window* of a particular task in a group is a time interval in which we estimate that a task should execute in order to ensure the correct precedence needed among group members and to allow parallel scheduling. The beginning point and the ending point of the interval are called the *pseudo ready time* and the *pseudo deadline*, respectively. The pseudo ready time is the earliest time that a task can start and the pseudo deadline is the latest time by which the task should finish. It does not matter where in the interval it is scheduled to start as long as it starts after the pseudo ready time and finishes before the pseudo deadline. To assign scheduling windows to tasks in a group, the originating node estimates a *group ready time* and uses a *time-slicing* scheme, in conjunction with a number of heuristic rules, to calculate a pseudo ready time and a pseudo deadline for each task. The

calculation of these pseudo time constraints takes into account the precedence constraints, the computation time requirements of tasks, as well as the current state of the network.

5.1.1 Estimating Group Ready Time

The *group ready time* of a group is the time at which the entire group is guaranteed and activated. Tasks will be scheduled to run between an estimated group ready time and the group deadline. The group ready time is not known until the algorithm completes, but some estimate of it is required to begin the algorithm. The estimated group ready time equals the current time plus an estimated scheduling delay which, for each new group, is calculated as a function of the size of the group and the delay observed for groups that arrived in a recent window,

$$EST(Scheduling\_delay) = New\_size \times \frac{Acc\_delay}{Acc\_size}$$

where New_size is the size of the new group; Acc_delay and Acc_size are the accumulated delay for groups that arrived in the recent window and the accumulated size of the groups, respectively.

If the estimated group ready time elapses before the algorithm completes, the algorithm invokes an adjustment phase to adjust the group ready time and the scheduling windows of tasks in the group, if possible. The adjustment algorithm is discussed in Section 5.4.

5.1.2 The Time-Slicing Scheme

The scheme for assigning scheduling windows is complicated and involves two notions: pseudo deadlines and time slices.

First, the pseudo ready time of all beginning tasks is set to be the estimated group ready time. The pseudo ready time of a descendant task j is then computed to be

$$PR_j = \max_i [PD_i + Comm(i,j)]$$

where PD_i is the pseudo deadline of a predecessor i; and $Comm(i,j)$ is the communication time between tasks i and j.

To calculate pseudo deadlines for tasks, the scheme we use considers a set of *parallel tasks* at a time. Parallel tasks are those whose pseudo ready times have been determined and can potentially run in parallel. The beginning tasks form the first set of parallel tasks to be considered. The calculation of pseudo deadlines are based on the following considerations:

1. If the computation time of a task is large (small), the scheduling window for the task should be large (small),

2. If parallel tasks have the same scheduling window and if the tasks are sent to the same node, they will compete for the processing power on the node within the window. If the window of the tasks is small, then it would be difficult to schedule all the tasks in the window on the node. Therefore, to reduce the potential scheduling conflicts that may occur in distributed scheduling, we assign a large window to a task, if the number of tasks that may run in parallel with the task is large, and vice versa.

3. If the surplus processing power of a node to which a task is sent is small, the scheduling window of the task should be large, and vice versa. However, because of the uncertainty about the destination node at this point in the algorithm, instead of the surplus of a specific node, we use the average surplus among all nodes with sufficient surplus to take part in distributed scheduling. A node is considered to have sufficient surplus to take part in distributed scheduling if its estimated surplus between the estimated group ready time and the group deadline is greater than the threshold NSF.

Each task in a group that can execute in parallel can have a different pseudo ready time and computation time. Therefore, the number of tasks that can potentially run in parallel at different points in time can vary. To calculate pseudo deadlines which incorporate the above heuristics, we consider a single *time slice* at a time. Since the general scheme for determining these time slices, and subsequently the pseudo deadlines, is complicated, we present a step by step example using the sample group.

We assume that, for the sample group, the estimated group ready time and the average surplus among nodes with sufficient surplus are 100 and 0.5, respectively. The precedence graph of the sample group and the scheduling windows calculated for the group are shown in Figure 2. Each scheduling window is delineated by a pseudo ready time PR_i and a pseudo deadline PD_i. The scheduling windows are constructed from six time slices, $S_1 = (100, 120), S_2 = (120, 125), S_3 = (125, 155), S_4 = (155, 160), S_5 = (165, 185)$, and $S_6 = (185, 205)$. Now we explain how these time slices were determined.

In the sample group, initially, only T_1 can run, so the pseudo deadline of T_1 is calculated as, $PD_1 = PR_1 + PTN \times C_1/Ave\_Node\_Surplus = 100 + 1 \times 10/0.5 = 120$, where PTN is the number of tasks that can potentially run in the slice and the $Ave\_Node\_Surplus$ is the average surplus among all nodes with sufficient surplus. The scheduling window of T_1 is considered as the first time slice, S_1. This slice accounts for the total amount of the computation time of T_1. After T_1, both T_2 and T_3 can potentially run in parallel. However, according to the policies of task clustering (see Section 5.2), T_2 is clustered with T_1, but T_3 is not. Therefore, $PR_2 = PD_1 = 120$ and $PR_3 = PD_1 + Comm(1, 3) = 120 + 5 = 125$. Since between PR_2 and PR_3, only T_2 can run and because, after PR_3, the number of parallel tasks is different, we assign the time slice (PR_2, PR_3) to the scheduling window of T_2. This slice is denoted as S_2. The size of this slice is $\Delta S_2 = PR_3 - PR_2 = 5$.

The interrelationship between the size of a time slice, ΔS_i, and the amount of computation time accounted for by the slice, ΔC_{Si}, is determined by the following equation:

$$\Delta S_i = PTN \times \Delta C_{Si}/Ave\_Node\_Surplus.$$

According to this equation, the computation time accounted for T_2 by this slice is,

$$\Delta C_{S2} = \Delta S_2 \times Ave\_Node\_Surplus/PTN = 5 \times 0.5/1 = 2.5$$

The remaining computation time of T_2 is $C_2 - \Delta C_{S2} = 10 - 2.5 = 7.5$. After PR_3, both T_2 and T_3 can potentially run in parallel. Since the remaining computation time of T_2 is less than $C_3 = 10$, T_2 can potentially finish before T_3. Therefore, we assign a time slice, S_3, to account for the remaining computation time of T_2. So, $\Delta C_{S3} = 7.5$. We assign this slice to the scheduling window of both T_2 and T_3. Since in this slice, both T_2 and T_3 can potentially run (i.e., $PTN = 2$), the size of this slice is calculated as

$\Delta S_3 = \Delta C_{S3} \times PTN/Ave\_Node\_Surplus = 7.5 \times 2/0.5 = 30$. Now the pseudo deadline of T_2 can be determined as

$$PD_2 = PR_2 + \Delta S_2 + \Delta S_3 = 120 + 5 + 30 = 155$$

The size of slices 4, 5, and 6 as well as PD_3, PD_4, PD_5 are determined in the same way as described above.

The computation of pseudo deadlines is the most expensive part in assigning scheduling windows. The time complexity of this scheme is $O(NSM^2)$, where N is the number of tasks in the group, S is the maximum number of slices that a task might have, and M is the maximum number of tasks that may run in parallel. If S and M are small compared to N, then the cost is linear in the size of the group. We are currently investigating a scheme where the pseudo deadlines are computed a priori, and consequently, independently of the current state of the network.

5.1.3 The Linear Modification Scheme

Since the scheme described above does not specifically take the group deadline into account, the scheduling windows may extend beyond the group deadline or may be less than the group deadline. If it is longer, the scheduling windows must be shortened to meet the deadline. If it is shorter, to utilize the extra time, the scheduling windows should be expanded. To keep the overhead low, we adopt a simple linear modification scheme described below.

The pseudo deadline of each task is extended (shortened) by an amount of time proportional to the total amount of time to which the scheduling windows can be expanded (must be shortened),

$$PD_i' = EGR + (PD_i - EGR) \times \frac{GD - EGR}{Max\_PD - EGR}$$

where PD_i is the original pseudo deadline for task i; Max_PD is the largest pseudo deadline among tasks in the group; and EGR and GD are the estimated group ready time and the group deadline, respectively.

The pseudo ready time of a task i is modified as follows:

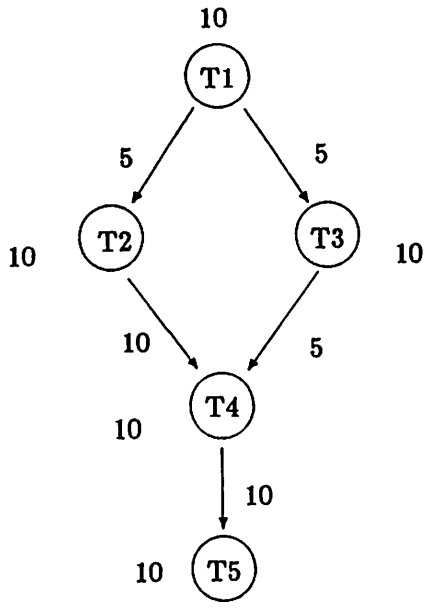
$$PR_i' = \max_j [PD_j' + Comm(j,i)]$$

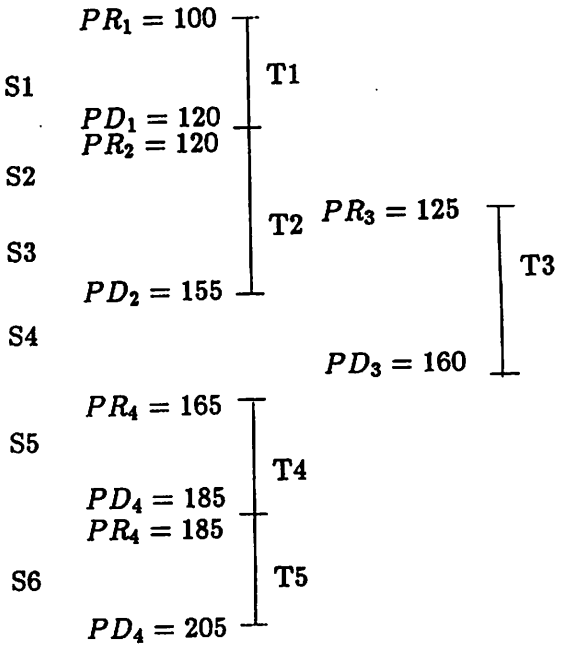
where PD_j' and $Comm(j,i)$ are the modified deadline of a predecessor j and the communication time between tasks j and i, respectively.

The scheduling windows of the sample group are expanded as shown in Table 1.

Note that, for a group of tasks with specified characteristics (i.e., precedence relations, computation time, and deadline), the estimated scheduling delay has strong impact on the scheduling windows. As the estimated delay increases, the size of the scheduling windows decreases, and vice versa. Since scheduling tasks in small scheduling windows is difficult, unnecessary scheduling overheads may be incurred if the windows are too small. To reduce overheads, the originating node checks the size of the scheduling window before distributing the tasks. If the size of the scheduling window for one or more tasks in the group is smaller than the task's computation time multiplied by a threshold SWF, the *Scheduling Window Fraction*, a tunable parameter, the windows are regarded as being too small. When such

EGR = 100
Ave. Node Surplus = 0.5
GD = 300
MCS = 2





(a) Precedence graph

(b) Scheduling windows.

Figure 2: The precedence graph and the scheduling windows of the sample group.

Task	PR	PD
T_1	100	138
T_2	138	219
T_3	143	224
T_4	229	262
T_5	262	300

Table 1: Expanded scheduling windows.

a situation occurs, the originating node aborts the group without attempting to distribute the group.

5.2 Clustering Tasks

To improve the schedulability of a task group, the originating node partitions a group into clusters, where tasks within a cluster have large inter–task communication costs. Thus by forcing tasks within a cluster to be scheduled to execute on the same node, these communication costs are eliminated. The originating node uses the following heuristics to extend the basic scheme described in Section 5.1 to determine the pattern of clustering (and to reduce the pseudo ready times):

- Let the *constraining predecessor* of a task be the predecessor whose enabling message is the last to arrive at the task. If a task is to be clustered with one or more of its predecessors, it must be clustered with the constraining predecessor. Otherwise, its pseudo ready time may not decrease.

- To reduce communication time between tasks, as many tasks as possible should be clustered with their constraining predecessor. However, if the size of a cluster grows too large, the probability of the cluster being scheduled at one node becomes low. Therefore, the size of clusters is limited to a threshold, MCS, the *Maximum Cluster Size*, a tunable parameter.

- If two tasks have the same constraining predecessor and if the size limitation prevents both tasks from being clustered with the predecessor, to reduce the effect of a long path, the one with higher *criticalness* is clustered with the predecessor. The criticalness of a task i is defined to be the length (in terms of time) of the longest path between the task and an ending task, i.e.,

$$Criticalness(i) = MAX(\sum_{j,k \in Path\ P} C_j + Comm(j,k))$$

  where path P is any path between task i and an ending task; C_j is the computation time of any task j on the path; and $Comm(j,k)$ is the worst case communication time between any two tasks j and k on the path. For the sample group, the criticalness of tasks is shown in Table 2.

Task	1	2	3	4	5
Criticalness	70	55	50	35	15

Table 2: The criticalness of tasks in the sample group.

- Let a *simple successor* be a successor with just one predecessor. If a task has a simple successor and the communication time between the tasks is greater than that between the task and its predecessors, then clustering the task with the simple successor is likely to reduce more communication time than clustering the task with its constraining predecessor. Therefore, the task should not be clustered with its constraining predecessor unless the resulting cluster size is still less than MCS.

The information about the criticalness of tasks and whether tasks have simple successors can be determined *statically* for a group by a straightforward topological search scheme. The time complexity of such a scheme is $O(N + E)$ time, where N are E are the number of nodes and the edges in the precedence graph of the group.

For the sample group, we assume that MCS equals 2. In the sample group, two clusters are formed, namely, $\{T_1, T_2\}$ and $\{T_4, T_5\}$. The constraining predecessor of both T_2 and T_3 is T_1. Because the criticalness of T_2 is greater than that of T_3 and because we limit the cluster size to 2, only T_2 is clustered with T_1. Also, because the communication time between T_4 and T_5 is greater than those between T_4 and T_2 or T_3, T_4 is clustered with T_5.

5.3 Scheduling Clusters Across the Network

After the originating node partitions a group of tasks into clusters, it decides which clusters can be scheduled locally and which clusters must be sent to other nodes. The node attempts to schedule as many clusters as possible. This step potentially reduces the number of clusters that must be sent out and hence reduces the overhead involved in the distributed scheduling. The clusters are processed in a *top-down* order: clusters that contain the beginning tasks are processed first; then, clusters that contain the descendants are processed. If a cluster cannot be scheduled locally, the originating node immediately invokes a scheme that combines focussed addressing and bidding [RS84] to decide where the cluster should be sent. This combined scheme is described below.

5.3.1 Focussed Addressing

The originating node uses surplus information about other nodes to decide which nodes have sufficient surplus to execute clusters that are not scheduled locally. Because the surplus information is periodically exchanged between nodes, at the time this information is used by a node, the amount of available surplus at each node is likely to be different. But if a node has sufficient surplus (larger than a threshold FP) needed to execute a cluster, the node should have a high probability of scheduling the cluster and so the originating node can directly send the cluster to that node. Such a node is called a *focussed node* for the cluster. More precisely, a node with surplus SR in the recent window is considered to be a

focussed node for a cluster, TC, if

$$\frac{SR \times \sum_{i \in TC}(PD_i - PR_i)}{Window\_size} \geq FP \times \sum_{i \in TC} C_i$$

where $Window\_size$ is the size of the moving window in which the surplus information is maintained.

If multiple focussed nodes are available, the originating node sends the cluster to the one with the greatest surplus. After the originating node sends a cluster to a node, it reduces the surplus of the node in the recent window by the amount of computation time required by the cluster. In this way, the originating node takes into account the clusters previously sent to different nodes and avoids sending too many clusters to the same node unless the node indeed has sufficient surplus for the clusters.

5.3.2 Bidding

For the clusters for which focussed nodes are not available, the originating node initiates a bidding process by sending *Request For Bid* messages to remote nodes and attempts to find a bidder node with sufficient surplus for each cluster. Each bidder node returns a bid to the originating node indicating its overall surplus processing power between the estimated group ready time and the group deadline. Based on this information, the originating node can send more than one cluster to a bidding node if that node happens to have the necessary surplus. More specifically, a bidder node with a surplus SR between the estimated group ready time, EGR, and the group deadline, GD, is a *good* bidder node for a cluster TC if,

$$\frac{SR \times \sum_{i \in TC}(PD_i - PR_i)}{GD - EGR} \geq \sum_{i \in TC} C_i$$

If multiple good bidder nodes are available, the originating node sends a cluster to the one with the greatest surplus. It also reduces the surplus of the node by the amount of computation time required by the cluster to reflect the fact that the cluster may be scheduled on the node and the surplus of the node will decrease.

After the originating node finishes creating clusters, then local and distributed scheduling are invoked in parallel. This step potentially reduces the delay involved in distributed scheduling thereby increasing the probability of scheduling a group. If distributed scheduling uses bidding for the cluster and that cluster is scheduled locally, then any returning bids are discarded.

Response to Cluster Transfer. Transferred clusters may or may not be scheduled at a receiving node. If a transferred cluster is scheduled on a node, the node sends a *confirmation* message to the originating node. If the originating node receives confirmation from all the nodes to which clusters in a group were sent, it guarantees and activates the group. However, if a cluster is sent to a focussed node and is not scheduled, it is sent to a node offering a good bid if one is available. To facilitate the latter decision and to reduce the scheduling delay, when the originating node initiates the bidding process for a cluster, the identity of focussed nodes for that cluster is sent to bidder nodes, and when a bidder node issues a bid, it sends a copy of the message to the focussed node. If a cluster is sent to a bidder node and is not scheduled, to reduce communication delay, the bidder

node invokes the adjustment algorithm (see next subsection) and attempts to reschedule the cluster locally.

5.4 Adjusting Scheduling Windows for Improving Schedulability

The scheduling windows[2] of tasks in a group need to be adjusted if the estimated group ready time elapses before the group is guaranteed or if a node finds that a cluster cannot be scheduled locally and no other node is available to schedule the cluster.

In the former case, the originating node attempts to extend the group ready time by adjusting the scheduling windows (extending the pseudo ready time and the pseudo deadline) of the beginning tasks as well as a sufficient number of their descendants. If the whole group can be scheduled before the extended group ready time, the group can still be guaranteed.

In the latter case, for each task that cannot be scheduled, a node attempts to adjust the scheduling windows (to extend the pseudo ready time and the pseudo deadline) on a range of successors in order to extend the pseudo deadline of the task in question. If the tasks can be scheduled with the extended pseudo deadlines, then the cluster is still schedulable. Otherwise, the group is aborted. Details of this adjustment algorithm are given below. To simplify the presentation, we assume that the adjustment algorithm is invoked to extend the pseudo deadline of a task.

To extend the pseudo deadline of a task, a node sends an *adjust request* message to each node processing a successor of the task. The request message carries an *Adjustment Target Time* which specifies the range of successors whose scheduling windows are to be adjusted. The request message is passed to the successors of each successor until it reaches a recipient task whose pseudo deadline is greater that the target time. The adjustment target time is computed as

$$Adjustment\_Target\_Time = PD + ATTR \times (GD - EGR)$$

where PD is the original pseudo deadline of the task which initiates the request and $ATTR$, the *Adjustment Target Time Ratio*, is a fraction and a tunable parameter. If the adjustment target time ratio is large, a task can request the adjustment of the scheduling windows on a large number of descendants thereby extending its pseudo deadline to a later time and increasing its own schedulability.

The adjustment of scheduling windows starts at the ending task of the specified range and is carried out in an order opposite to that of passing the requests. Each node attempts to extend the pseudo deadline and the pseudo ready time for each requested task that resides locally: The extended pseudo deadline of a task i is computed as,

$$EPD_i = \min_j [EPR_j - Comm(i,j)]$$

where EPR_j and $Comm(i,j)$ are the extended pseudo ready time of a successor j and the communication time between i and j, respectively. Note that the pseudo deadline of the ending tasks in the specified adjustment range is not extended. Each task is scheduled with respect to the extended pseudo deadline (or rescheduled if it has been previously scheduled). If it is scheduled, the pseudo ready time of the task is extended to be the latest start time

[2]Recall that scheduling windows are just estimates used to facilitate parallel processing of the task group.

of the task and this information is sent to the (intermediate) requesting predecessor. The predecessors are then scheduled with respect to their extended pseudo deadlines. The information on the extended pseudo ready time of the predecessors is sent to the requesting nodes where the predecessors of the predecessors exist. This process is repeated until the pseudo deadline of the original requesting task is extended and the attempt of scheduling the task with respect to the extended pseudo deadline is accomplished. If an intermediate task cannot be scheduled, the node where the task resides initiates an adjustment request for the task itself. Note that multiple adjustment requests may be initiated simultaneously in a group. If a task receives two requests, it only passes the one with greater target time. However, after its extended pseudo ready time is computed, it acknowledges both requests.

To demonstrate the use of the adjustment scheme, we describe a scenario of distributed scheduling for the sample group. When reading this example, the readers should refer to Figure 2 and Table 1. As mentioned before, the sample group has three clusters, i.e., $\{T_1, T_2\}, \{T_3\}$, and $\{T_4, T_5\}$. Suppose the first cluster $\{T_1, T_2\}$ is scheduled locally at node 1, the second cluster $\{T_3\}$ is sent to a bidder node, node 2, and the third cluster $\{T_4, T_5\}$ is sent to a focussed node, node 3. Suppose the second cluster is not scheduled, but the third cluster is scheduled. Since node 2 is a bidder node, it invokes the adjustment scheme for the cluster. Suppose $ATTR = 0.25$. Then, node 2 sends a request message to node 3. The message specifies an adjustment target time which equals, $PD_3 + ATTR \times (GD - EGR) = 224 + 0.25 \times (300 - 100) = 274$. Since the target time is greater than $PD_4 = 262$, node 3 passes the request to T_5 which is also on node 3. Because the adjustment target time is less than $PD_5 = 300$, node 3 starts the actual adjustment beginning with T_5. Because T_5 has been previously scheduled, its pseudo ready time is simply extended to its latest start time which is 290. So, the pseudo deadline of T_4 is extended to 290 and T_4 is rescheduled. Suppose the new latest start time of T_4 is 270. Then, the pseudo ready time of T_4 is extended to 270, and this information is sent to node 2. Finally, the pseudo deadline of T_3 is extended to $EPD_4 - Comm(3,4) = 270 - 5 = 265$ and the task is scheduled.

The cost of the adjustment scheme is analyzed as follows. First, we consider the communication cost: When a task initiates an adjustment process, its request message is passed to a subset of successors which form a tree in the group where the root of the tree is the initiating task and the depth of the tree is specified by the adjustment target time. Also, when a successor acknowledges an adjustment request, it only sends messages to its predecessors in the tree. Therefore, the number of messages involved in an adjustment process is proportional to the number of edges in the tree, $O(E)$. Second, we consider processing cost: Clearly, in the worst case, except the leaves, all the other tasks in the tree have to be rescheduled to find a new latest start time. Therefore, the additional processing cost due to adjustment is proportional to the number of tasks in the tree, $O(N)$. In summary, the cost of the adjustment scheme is $O(N + E)$. In Section 7 where we discuss simulation results, we will examine the impact on the system performance by using different adjustment ranges.

6 Evaluation Approach

In the previous sections, we have described three atomic guarantee algorithms for the group scheduling problem, namely, the sequential algorithm, the polling algorithm, and the partition algorithm. It is very difficult (if not impossible) to develop a tractable analytical model

for hard real-time distributed scheduling problems. Therefore, we evaluate and compare these distributed guarantee algorithms by simulation. However, the algorithms were implemented, their execution times measured and this cost was then used as one of the inputs to the simulation model. We evaluate these algorithms under a wide range of system conditions and for different types of groups. For example, we test different communication delays from 0 to 10 milliseconds per packet, different laxities from a few hundred milliseconds to a few seconds, different sizes of groups from 1 to 30, different types of precedence constraints (including precedence chains, precedence trees, and arbitrary precedence constraints), and different mixtures (distributions) of group sizes. We believe that the range of the test conditions is large enough to cover future applications of many real systems. In this section, we describe a simulation model for the guarantee algorithms. We also present two baseline algorithms and the performance metric for the analysis of these guarantee algorithms. The results of the extensive simulation studies are described in Section 7.

6.1 The Simulation Model

The simulation program is written in GPSS and Ada and it simulates a number of nodes interconnected by a communication network. For a particular test, the nodes run either the sequential, the polling, and the partition algorithm. The partition algorithm contains a number of system tunable parameters. This makes the algorithm flexible, but causes difficulty in choosing and maintaining good values for these parameters. In practice it was not too difficult to tune this set of parameters, and the values chosen seem to be fairly robust. In addition, the simulation model consists of a number of simulation parameters. The nominal values for the major simulation parameters are shown in Table 3. In the simulation studies described in the next section, unless otherwise mentioned, the parameters are set according to the nominal values. All parameters of the model are easily changed to facilitate evaluation of a particular guarantee algorithm and the comparison of different algorithms. We now describe how to create groups as input to the simulation, how the communication network is modeled, and how the scheduling cost is incorporated.

6.1.1 Group Generation

Task groups are nonperiodic and are generated randomly on each node in the network. Because actual systems are not available, the arrival pattern of the task groups as well as the characteristics of the groups are created based on a probablistic model instead of actual workloads. The characteristics of each group is specified by a number of parameters. The value of the parameters, the arrival rate and the precedence structure of groups are generated from random number generators of the simulation model as described below.

The number of tasks in a group (called the *group size*) is generated from a probability distribution with an average $\mathcal{G}$. We test three distributions, namely, normal distribution, exponential distribution, and uniform distribution. The standard deviation of the normal distribution is set to be 20% of the average.

The following parameters of a task group are normally distributed: 1) The computation time of each task, with an average C, 2) the communication time between tasks in a group, with an average M, and 3) the size of each task (i.e., the number of packets), with an average S. The standard deviation of each parameter is set to be 20% of the mean of the

Average group size	6	
Task computation time	100	(milliseconds)
Execution time ratio	0.7	
Task size	10	(packets)
Communication time ratio	0.1	
Laxity	2	(seconds)
Network load	6	
Balance factor	0.7	
Layer size ratio	0.3	
Message delay	4	(milliseconds/packet)
Guarantee unit cost	0.3	(millisecond)
Preprocessing unit cost	1.2	(milliseconds)
Number of nodes	5	

Table 3: Nominal value of major simulation parameters.

parameter. C is set to be 100 milliseconds and is not changed between runs. It is used as a normalization factor for other parameters of groups. M is set to be a fraction of C. The fraction is called as the *communication time ratio.*

The deadline of a group is relative to the arrival time of the group. It is calculated as follows:

$$D = arrival\ time + \sum_i C_i + \sum_{i,j} Comm(i,j) + Laxity$$

where C_i is the computation time of any task i in the group; $Comm(i,j)$ is the communication time between any two tasks i and j on the longest path in the group; and *Laxity* is a simulation parameter. Note that, even if the laxity is zero, it is still possible to guarantee a group at the node where the group arrives.

The arrival rate of groups at a node is a function of a predetermined load factor of the node. We define the *load factor* of a node to be the *average group computation time* divided by the interarrival time of the groups arriving at the node. The average group computation time equals the average task computation time multiplied by the average group size. The load factor of different nodes is set to be increasing linearly from node 1 to 2, to 3, etc. Let $Load_1$ and $Load_i$ denote the load of node 1 and load of node i, respectively. Then,

$$Load_i = Load_1 \times R^{i-1}$$

where the proportionality constant R is called the *balance factor*. The sum of the load factors of all the nodes is called the *network load.* The network load represents the demands on the system imposed by the environment. The actual system load depends on the number of groups guaranteed in the system.

The precedence graphs of task groups (for input to the simulation model) are constructed as follows. Tasks in a group are divided into layers. Each *layer* is a subset of tasks which interact only with those in the adjacent layers. A task in a layer i can only be the predecessor of tasks in layer $i+1$ (lower layer) and the successor of those in layer $i-1$ (upper layer).

Note that tasks in the top layer (layer 1) do not have predecessors and those in the bottom layer do not have successors. The group is constructed starting from the top layer and proceeding to the bottom layer. For each layer, the number of tasks is chosen randomly according to a uniform distribution between one and a fraction of the group size. The fraction, called as the *layer size ratio*, is a simulation parameter. The number of layers in a group depends on the number of tasks actually chosen for each layer and the group size. The number of predecessors (successors) of a task is chosen randomly according to a uniform distribution between one and the number of tasks in the immediate upper (lower) layer.

The above strategy is used in order to simplify the construction of the precedence graph. However, it is general enough to generate most interesting types of precedence graph, including chain graphs, tree graphs, serial–parallel graphs, as well as arbitrary precedence graphs.

6.1.2 Communication Network

The network consists of five nodes interconnected by a shared–bus network as shown in Figure 3. Each node is connected to the shared bus by a link. Messages sent from one node to another pass through a sender's link, the shared bus, and a receiver's link. The links and the shared bus are used to model the queueing delay within nodes and the access delay to the shared bus. Only one message can occupy a link (the bus) at a time. When a message is in a link (the bus), if there is another one that needs to be sent through the same link (bus), the latter must wait until the first one relinquishes the link (bus). Messages to be sent through the same link (bus) are sent on a FCFS basis.

The net cost of transmitting a message from one node to another without delay is called the *transmission time*. We assume that the cost of processing messages within nodes is comparable to the cost of transmitting messages. Therefore, the time spent by a message in a sender's link, the shared bus, and a receiver's link is set to be one third of the total transmission time. The transmission time of messages is set to be proportional to the message size. The control messages (e.g., RFB and bid) consist of just one packet. The transmission time for such messages is called the *message delay* which is a simulation parameter. The transmission time for a task equals the message delay times the size of the task.

The messages exchanged between application tasks are delivered within constant time and are not subject to the delay described above. The worst case time for transmitting such messages are prespecified for a group.

6.1.3 Scheduling Overhead

We assume that a co–processor is available for executing system scheduling tasks and the scheduling cost can be off–loaded from the main processor. However, we do account for scheduling delay. The delay in scheduling a single task on a node is proportional to the number of tasks prescheduled on the node. The proportionality constant is called the *guarantee unit cost* which is a simulation parameter. In each guarantee algorithm, multiple attempts may be made to schedule a task on a node. The delay of every attempt is included in the simulation model. For different guarantee algorithms, the guarantee unit cost is the same. However, the total delay for scheduling each distributed group is different.

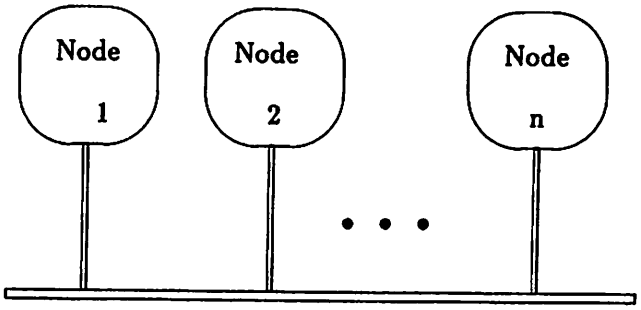


Figure 3: Communication network.

For the partition algorithm, the cost of partitioning a group and assigning scheduling windows is set to be proportional to the size of a group. The proportionality constant is called as the *preprocessing unit cost.*

In the simulation model, both the guarantee unit cost and the preprocessing unit cost are set by measuring the actual algorithm executing on a VAX-11/780 system.

It should be mentioned here that it is possible to off-load the preprocessing cost by performing the preprocessing scheme on each possible task group statically with respect to various possible system states, i.e., different combinations of node surplus and system scheduling delay, and keeping the partition information together with the group. When it becomes necessary to distribute a group, based on the current system state, a node selects a precalculated partition pattern for the group. In the simulation studies, we assume that the preprocessing is done on-line, and hence, include these costs in the scheduling overheads of every group.

6.2 Baseline Algorithms

To validate the performance of the three guarantee algorithms discussed in this paper, we compare these algorithms with two baseline algorithms: the perfect-information algorithm and the noncooperative algorithm.

6.2.1 Perfect-Information Algorithm

The first baseline algorithm is called the *perfect-information* algorithm. It assumes that each node has perfect state information about other nodes and incurs no delay to distribute and schedule tasks. This algorithm is similar to the polling algorithm without overheads: Both the communication delay and the processing delay are assumed to be zero. To guarantee a group, a node polls *every* node in the network and attempts to determine the latest start time for every task in the group. Because the network-wide latest start times of tasks are determined without delay, this algorithm should produce better results than the guarantee algorithms and provides an upper bound to the performance of the various atomic guarantee algorithms.

6.2.2 Non-Cooperative Algorithm

The second baseline algorithm is called the *non-cooperative* algorithm. It models worst case performance. This algorithm requires that each group be completely guaranteed at the node where the group originally arrives. If a group cannot be scheduled, it is aborted. No attempt is made to distribute the group. The local scheduling delay is accounted for in this algorithm.

6.3 Performance Metric and Statistics

We are particularly interested in the performance metric, referred to as the *system guarantee ratio*. The system guarantee ratio is defined as the total number of groups guaranteed versus the total number of groups that have arrived in the network.

Each simulation runs for 2,000,000 time units (milliseconds). It is long enough for the simulation model to approximately reach equilibrium state (i.e., the fluctuation in the guarantee ratio is less than 2%). To show the statistical validity of the simulation results, in next section, we calculate the 90% confidence interval for the tests shown in Sections 7.2 and 7.3. Given that these intervals are small and given simulation cost constraints, we do not calculate the confidence interval for all the tests. The results of the other tests are obtained from single, but sufficiently long runs.

7 Simulation Results

In this section, we describe the simulation results for the three atomic guarantee algorithms described in Sections 4 and 5. We compare the partition algorithm with the sequential algorithm and the polling algorithm under a wide range of system conditions and for groups of different characteristics. The results are validated against the baseline algorithms. Specific experiments were run to show need for the various components of the partition algorithm. We show that all components of the algorithm contribute to its overall better performance. Due to space limitations these results are not shown here (see [Che87]. The experiments we consider in this paper are:

1. Effect of different communication costs
2. Effect of different laxities
3. Effect of different inter-task communication times
4. Effect of different group sizes: arbitrary precedence graphs
5. Effect of different group sizes: precedence trees
6. Effect of different group sizes: precedence chains

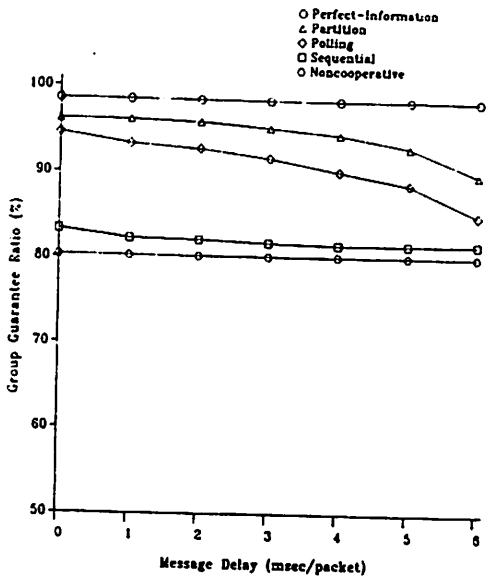


Figure 4: Effect of different communication costs (laxity = 1 second, average group size = 6, network load = 6).

7.1 Effect of Different Communication Costs

In this subsection, we examine the effect of different communication costs. The message delay is varied from 0 to 6 milliseconds, with increments of 1 millisecond. We test two cases of laxities, namely, 1 and 1.5 seconds. The average group size is set to 6. The simulation results are shown in Figures 4 and 5. Note that the communication cost has no effect on the perfect-information algorithm nor on the noncooperative algorithm.

From Figures 4 and 5, we observe the following:

- As the message delay increases, the guarantee ratio of the partition algorithm and the alternative guarantee algorithms decreases. For example, for a laxity of 1 second, as the message delay increases from 0 to 6 milliseconds, the partition, the polling, and the sequential algorithm, degrade by 6.1%, 9.3%, and 1.4%, respectively. This effect is obvious, because, as the communication cost increases, the delay involved in distributed scheduling increases, so it is more difficult to guarantee distributed groups.

- The communication cost has slightly greater effect on the polling algorithm than on the partition algorithm. For example, for a laxity of 1 second, as the message delay increases from 0 to 6 milliseconds, the polling algorithm degrades by 9.3%, but the partition algorithm degrades only by 6.1%; the difference in the guarantee ratio of the two algorithms increases from 1.6% to 4.8%. This result indicates that the communication cost incurred by the polling algorithm is slightly greater than that of the partition algorithm. Recall that, in the polling algorithm, most of the communication occurs across the network, but in the partition algorithm, because of task clustering, a significant amount of the communication occurs within nodes.

- In the tested range of communication costs, the performance of the sequential algo-

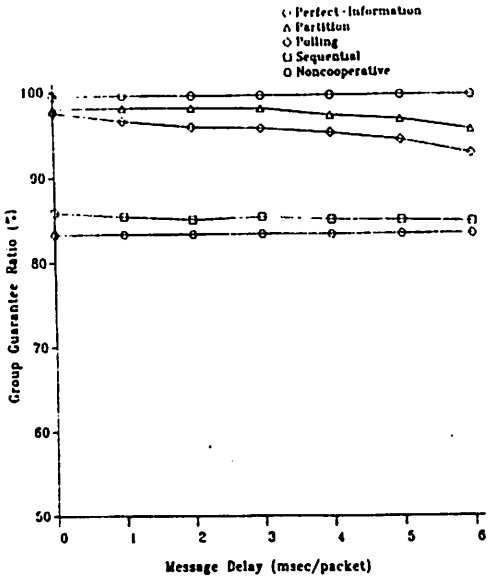


Figure 5: Effect of different communication costs (laxity = 1.5 second, average group size = 6, network load = 6).

rithm and the noncooperative algorithm is about the same. For all the tested cases, the difference between the two algorithms is less than 3%. Also, for all the tested cases, the partition algorithm performs significantly better than the sequential algorithm. For example, for a laxity of 1 second, as the message delay increases from 0 to 6 milliseconds, the difference in the guarantee ratio between the two algorithms decreases only from 12.8% to 8.2%. Because of this result, we conclude that the partition algorithm performs much better than the sequential algorithm. Therefore, in the succeeding simulation studies, we show the result on the sequential algorithm without discussion.

7.2 Effect of Different Laxities

In this subsection, we compare the performance of different guarantee algorithm under different laxities. The laxity is increased from 0.5 to 3 seconds, with increments of 0.5 seconds. The average group size is set to be 6. The simulation results are shown in Figure 6. The confidence intervals are shown in Table 4.

From Figure 6, we observe the following:

- As the laxity increases, the performance of all the algorithms improves. For example, as the laxity increases from 0.5 to 3 seconds, the guarantee ratio of the partition algorithm and the polling algorithm improves by 18.4% and 12.2%, respectively. This effect is obvious, because, as the laxity increases, it is easier to guarantee groups both locally and across the network.

- When the laxity is small, the partition algorithm performs better than the polling algorithm. For example, when the laxity is 0.5 second, the difference in the guarantee

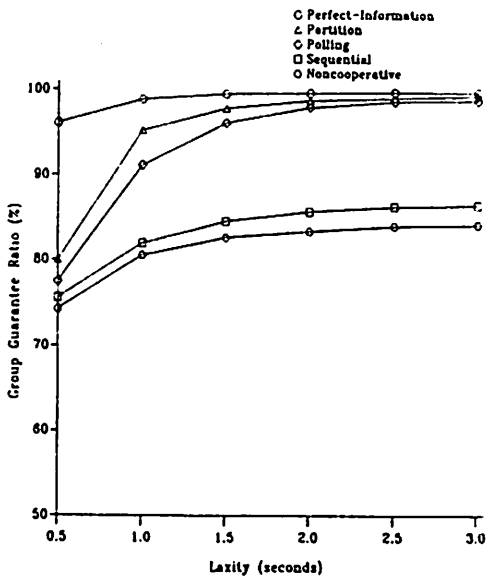


Figure 6: Effect of different laxities (average group size = 6, network load = 6).

ratio of the two algorithms is 4.6%. This result indicates that the overall scheduling delay of the partition algorithm is smaller than that of the polling algorithm, so the partition algorithm performs better than the polling algorithm for groups of moderate sizes and small laxities.

- When the laxity is moderate, the partition algorithm still performs better than the polling algorithm. For example, when the laxity is 1 second, the guarantee ratio of the partition algorithm is greater than that of the polling algorithm by 4.3%. This result indicates that the advantages of the partition algorithm are significant in a wide range of laxities.

- When the laxity is large, the difference between the partition algorithm and the polling algorithm is small. For example, when the laxity is greater than 2 seconds, the difference between the two algorithms is less than 1%. This indicates that, for groups with large laxities, both algorithms can perform well, and it is not essential to use a sophisticated algorithm.

- In the tested range of laxities, the partition algorithm performs substantially better than the noncooperative algorithm. For all the tested cases, the difference between the two algorithms is about 12%. This indicates that, in a wide range of laxities, the system can benefit substantially from distributed scheduling.

7.3 Effect of Different Inter-Task Communication Times

In this subsection, we examine the effect of different inter-task communication times on the guarantee ratio of the partition algorithm and the polling algorithm. The partition

Laxity (seconds)	Algorithm				
	Perfect-Info.	Partition	Polling	Sequential	Noncoop.
0.5	96.1 ± 0.44	80.0 ± 1.96	77.5 ± 1.21	75.6 ± 0.67	74.3 ± 0.62
1.0	98.9 ± 0.25	95.3 ± 0.40	91.2 ± 0.96	82.0 ± 0.72	80.6 ± 1.23
1.5	99.6 ± 0.13	97.9 ± 0.48	96.2 ± 0.65	84.6 ± 0.56	82.7 ± 1.01
2.0	99.8 ± 0.15	98.9 ± 0.36	98.1 ± 0.26	85.8 ± 0.83	83.5 ± 1.02
2.5	99.9 ± 0.05	99.2 ± 0.34	98.8 ± 0.29	86.4 ± 0.58	84.1 ± 1.09
3.0	99.9 ± 0.01	99.5 ± 0.15	98.9 ± 0.53	86.6 ± 0.43	84.3 ± 0.64

Table 4: The 90% confidence intervals for Figure 6.

algorithm explicitly clusters tasks with each other if the tasks have large inter-task communication time. On the other hand, the polling algorithm does not explicitly cluster tasks on the same node, but it tends to schedule a task and its successors on the same node if such an attempt improves the latest start time of the task. In other words, the communication time between tasks is eliminated by implicit efforts. Note that, in the simulation model, the laxity of a group includes the communication time between tasks on the critical path in the group. Therefore, if tasks are scheduled on the same node, the leeway for scheduling tasks will be greater than if tasks are scheduled on different nodes.

In the simulation study, the inter-task communication time ratio is increased from 0.1 to 0.9, with increments of 0.2. The average inter-task communication time equals the inter-task communication time ratio multiplied by the average task computation time. For example, if the inter-task communication time ratio is 0.5, then the average inter-task communication time equals 50 milliseconds. We test two cases of network loads, namely, 6 and 7. The average group size and laxity are set to 10 and 1 second, respectively. We compare the algorithms under arbitrary precedence graphs, precedence (reverse) trees, and precedence chains. The simulation results are shown in Figures 8, 9, and 10, respectively. An example of a reverse tree is shown in Figure 7, where each task has multiple predecessors, but only one successor. The group has multiple beginning tasks, but only one ending task. For the arbitrary precedence graphs, we show the 90% confidence intervals of the results in Table 5.

From Figures 8, 9, and 10, we observe the following:

- Except for the case of the partition algorithm applied to precedence trees at network load 6 (Figure 9), the guarantee ratio of both the partition and polling algorithms increases significantly as the inter-task communication time ratio increases. For example, as the inter-task communication time ratio increases from 0.1 to 0.9, when the network load is 7, for arbitrary precedence graphs, the partition algorithm and the polling algorithm improve by 7.7% and by 6.5%, respectively; for precedence trees, the two algorithms improve by 9.9% and 4.5%, respectively; and for precedence chains, by 9.3% and 10.3%, respectively. These results are due to the extra leeway obtained by scheduling tasks on the same node. In the exceptional case of the partition algorithm indicated above, the improvement is only 1.8%. This occurs because the critical path of a tree group is smaller than other types of groups and the laxities are relatively

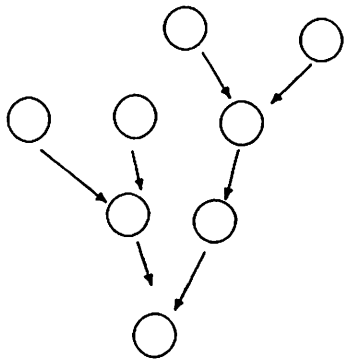


Figure 7: A reverse tree.

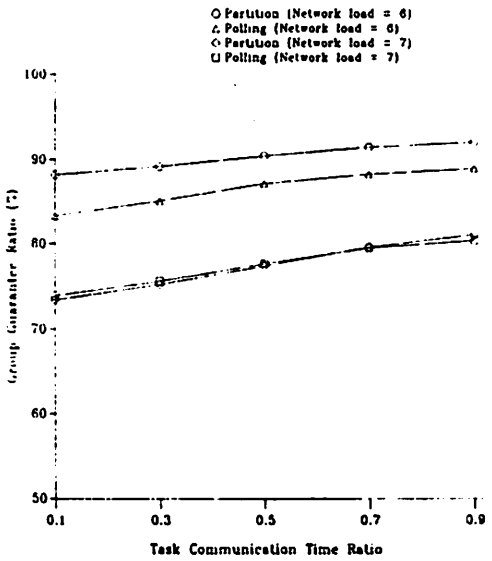


Figure 8: Effect of different inter-task communication times: arbitrary precedence graphs (average group size = 10, laxity = 1 second).

Task Communication Time Ratio	Partition Algorithm		Polling Algorithm	
	Network Load		Network Load	
	6	7	6	7
0.1	88.2 ±0.94	73.2 ±0.67	83.3 ±0.80	73.9 ±1.29
0.3	89.2 ±1.17	74.8 ±0.66	85.1 ±1.07	75.6 ±1.17
0.5	90.5 ±0.86	76.7 ±0.70	87.2 ±0.81	77.7 ±1.20
0.7	92.1 ±0.97	79.2 ±0.69	88.3 ±1.01	79.5 ±1.32
0.9	92.6 ±0.89	81.4 ±0.48	89.0 ±1.13	80.4 ±1.31

Table 5: The 90% confidence intervals for Figure 8.

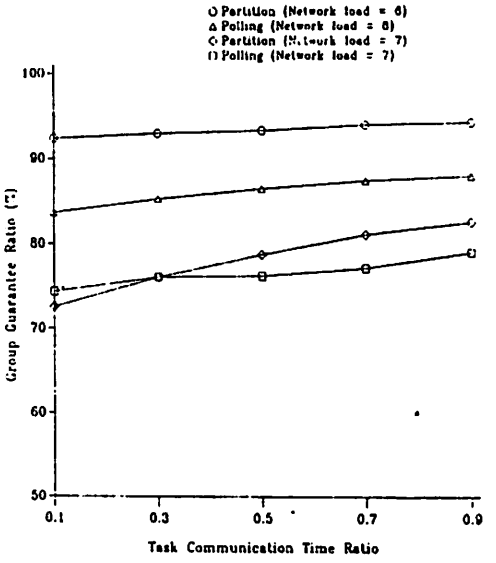


Figure 9: Effect of different inter-task communication times: precedence trees (average group size = 10, laxity = 1 second).

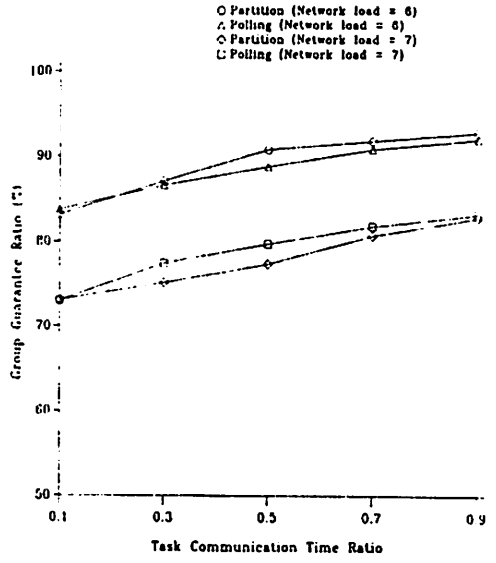


Figure 10: Effect of different inter-task communication times: precedence chains (average group size = 10, laxity = 1 second).

large for such groups and such network loads. In this case, the guarantee ratio is high (between 92% and 94%), so the extra leeway does not significantly improve the guarantee ratio.

- For moderate network loads, the improvement in the guarantee ratio for the case of precedence chains is greater than those for the case of precedence trees and arbitrary precedence graphs. For example, for network load 6 and for precedence chains, the improvement for the partition algorithm is 9.7%, but it is only 1.8% and 3.9% for arbitrary precedence graphs and precedence trees, respectively. This result occurs because the critical path in a chain group is longer than those of a group with arbitrary precedence graph or a precedence tree. Therefore, the effect of task clustering is larger for chain groups.

- For heavy network loads, the improvement in the guarantee ratio of the partition algorithm is about the same regardless of different types of precedence graphs. For example, for network load 7, the improvement of the algorithm is 7.7%, 9.9%, and 9.3% for arbitrary precedence graphs, precedence trees, and precedence chains, respectively. This is because, when the network load is heavy, the scheduling delay is large. Therefore, the laxity of groups become tight, and the extra leeway due to task clustering becomes important.

- Except for network load 6 and for arbitrary precedence graphs and precedence trees, the improvement in the guarantee ratio for the partition algorithm is either about the same or slightly greater than that of the polling algorithm. This result indicates that the scheme of task clustering in the partition algorithm is either as effective as or better than the implicit effort of the polling algorithm on clustering tasks. In the partition algorithm, clustering tasks is done with local preprocessing cost which is low compared to the polling cost in the polling algorithm.

7.4 Effect of Different Group Sizes: Arbitrary Precedence Graphs

In this subsection, we examine the effect of different sizes of groups. The purpose of this and the following two simulation studies is to compare the various algorithms for groups of different types of precedence structures. In this study, we test arbitrary precedence graphs constructed as described in Section 6.1. The average group size is increased from 4 to 20, with increments of 4. We test two cases of laxities, namely, 1.5 and 2 seconds. The simulation results are shown in Figure 11 and 12.

From Figures 11 and 12, we observe the following:

- When the group size increases, the guarantee ratio of all the algorithms decreases. For example, for laxity of 1.5 seconds, as the average group size increases from 4 to 20, the guarantee ratio of the partition and the polling algorithm decreases from 98.5% to 76.5% and from 97.8% to 77.3%, respectively. This result is obviously due to the following reasons: (1) As the average group size increases, the number of tasks involved in atomic guarantee increases and the scheduling delay increases, so it is more difficult to guarantee groups; (2) For a fixed laxity, as the average group size increases, the leeway for scheduling a group decreases.

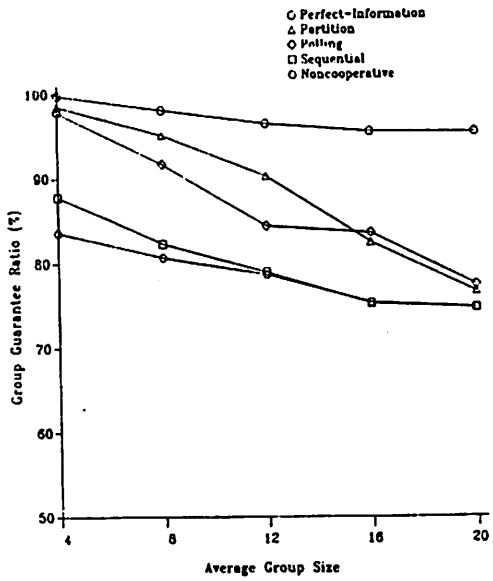


Figure 11: Effect of different group sizes: arbitrary precedence graphs (laxity = 1.5 seconds, network load = 6).

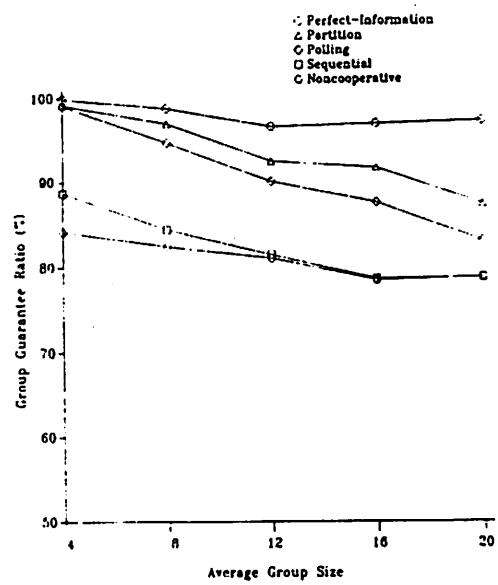


Figure 12: Effect of different group sizes: arbitrary precedence graphs (laxity = 2 seconds, network load = 6).

- When the average group size is small, the partition algorithm and the polling algorithm perform about the same. For example, for laxity of 1.5 seconds, when the average group size is 4, the difference in the guarantee ratio of the two algorithms is less than 1%. This occurs because the laxity is relatively large for such groups.

- When the group size is moderate, the partition algorithm performs better than the polling algorithm. For example, when the average group size is 12, for laxity of 1.5 and 2 seconds, the guarantee ratio of the partition algorithm is greater than that of the polling algorithm by 6 %and 2.4%, respectively. This result indicates that, for groups of moderate sizes and moderate or large laxities, the partition algorithm performs better than the polling algorithm.

- For large groups with moderate laxity, the partition algorithm performs about the same as the polling algorithm. For example, for laxity of 1.5 seconds, when the average group size is greater than 16, the difference in the guarantee ratio of the two algorithms is only about 1%. The reason that the partition algorithm does not perform better than the polling algorithm is because it generates small scheduling windows due to the large group size and the moderate laxity.

- For large groups with large laxity, the partition algorithm performs better than the polling algorithm. For example, for a laxity of 2 seconds, when the average group size is 20, the partition algorithm performs better than the polling algorithm by 4.3%. This result occurs because, as in the previous case, the scheduling windows are small.

The above observations for arbitrary precedence graphs are summarized in Table 6(a) where we show which algorithm performs better under given conditions of laxity and group size.

7.5 Effect of Different Group Sizes: Precedence Trees

In this subsection, the precedence graphs of groups are constructed such as a *reverse tree* as described before in Section 7.3. Besides comparing with the polling algorithm and the baseline algorithms, we compare the partition algorithm with a different version of the sequential algorithm. When a node receives a tree group (subgroup), it attempts to schedule as many tasks in the tree (subtree) as possible in a reverse topological order, but if it finds a task that is not schedulable, instead of sending all the unscheduled tasks in the tree to another node, it only sends a subtree rooted at the task to another node. It continues its attempt to schedule tasks in other parts of the tree. In this way, the attempt to schedule tasks in different subtrees can be carried out in parallel at multiple nodes. In order to maintain the consistency of notation, we retain the name of *sequential* algorithm.

In the simulation study, the size of groups is normally distributed. The average group size is increased from 4 to 20, with increments of 4. We test two cases of laxities, i.e., 1.5 and 2 seconds. The simulation results are shown in Figure 13 and 14.

From Figure 13 and 14, we observe the following:

- For small trees, the guarantee ratio of the partition algorithm and the polling algorithm is about the same. This result is the same as that for arbitrary precedence graphs.

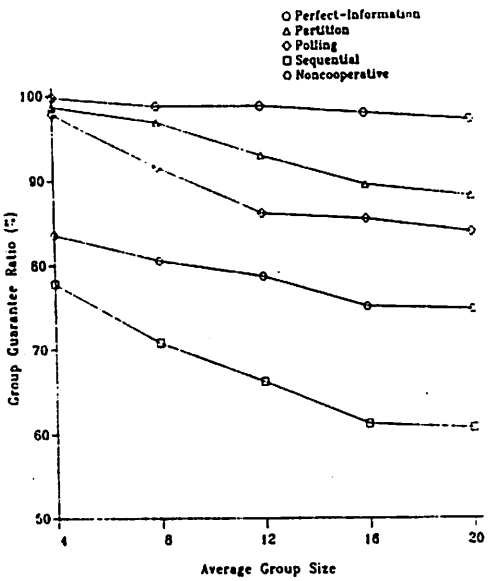


Figure 13: Effect of different group sizes: precedence trees (laxity = 1.5 seconds, network load = 6).

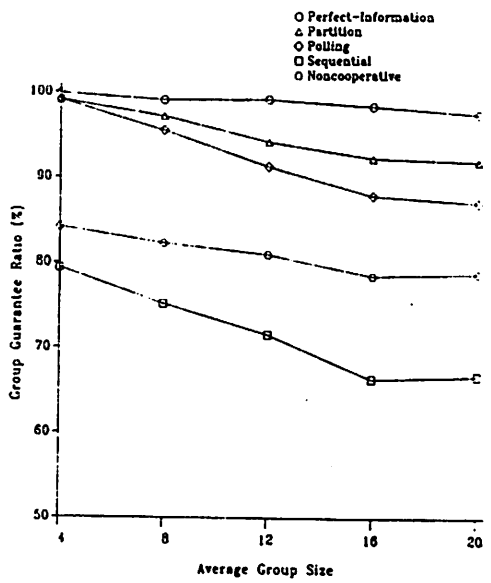


Figure 14: Effect of different group sizes: precedence trees (laxity = 2 seconds, network load = 6).

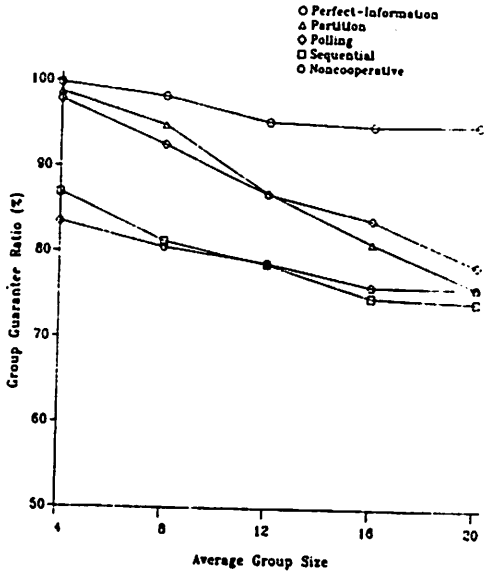


Figure 15: Effect of different group sizes: precedence chains (laxity = 1.5 seconds, network load = 6).

- However, for trees of moderate and large sizes, the guarantee ratio of the partition algorithm is significantly greater than that of the polling algorithm. For example, when the laxity is 1.5 seconds, for group sizes of 12 and 20, the partition algorithm performs better than the polling algorithm by 6.8% and 4.2%, respectively. The reason for the better performance of the partition algorithm for large trees is because the depth of the trees is short (the number of leaves in the trees is large), and therefore, the system can assign large scheduling windows to tasks. This result indicates that the partition algorithm works very well for precedence trees.
- In all tested cases, the partition algorithm performs substantially better than the sequential algorithm. This indicates that, for the sequential algorithm, even though different subtrees in a group can be processed at different nodes in parallel, because each task is scheduled on one node at a time, the same reason that causes poor performance of the original sequential algorithm is still applicable.

7.6 Effect of Different Group Sizes: Precedence Chains

In this subsection, the precedence graph of groups is constructed as a *chain*. Each task in a group has exactly one predecessor and one successor. We compare different guarantee algorithms for precedence chains of different sizes. The size of groups is normally distributed. The average group size is increased from 4 to 20, with increments of 4. We test two cases of laxities, namely, 1.5 and 2 seconds. The simulation results are shown in Figure 15 and 16.

From Figures 15 and 16, we observe the following:

- For small chains, the performance of the partition algorithm and the polling algorithm is about the same. This is the same as that for arbitrary precedence graphs and trees.

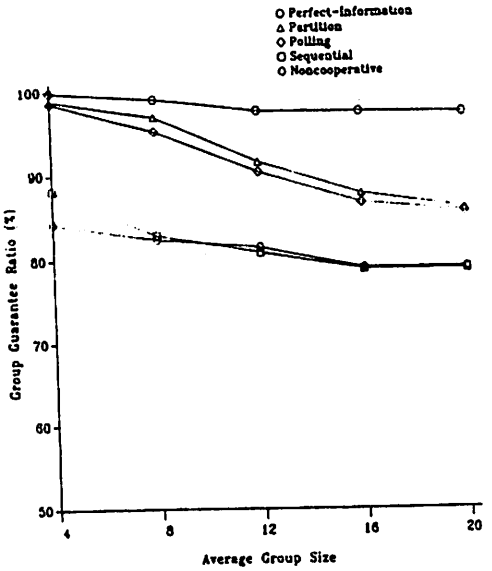


Figure 16: Effect of different group sizes: precedence chains (laxity = 2 seconds, network load = 6).

- However, for chains of moderate sizes, the partition algorithm only performs slightly better than the polling algorithm. For long chains with moderate laxity, the polling algorithm performs better than the partition algorithm. For example, when the average group size is 16, for laxity of 1.5 seconds, the guarantee ratio of the polling algorithm is greater than that of the partition algorithm by 3.2%. These results occur because, compared to other types of precedence graphs, the laxities for groups of chain-type precedence graphs is tight, since every task is on the critical path. Therefore, the scheduling windows for groups of moderate or large sizes are small and the performance of the partition algorithm degrades. This result indicates that, for precedence chains, the group size has a large effect on the partition algorithm.

8 Conclusions

We have developed an atomic guarantee algorithm that can handle arbitrary types of precedence constraints among tasks. This algorithm is based on a partition scheme. To reduce distributed scheduling delay, the algorithm divides tasks in a group into subgroups and distributes the subgroups across nodes to be scheduled. This scheme allows the system to schedule tasks in a group in parallel and increases the chance of scheduling a distributed group.

The simulation studies have shown that, compared to the alternative algorithms, the partition algorithm is effective under many conditions in dynamic hard real-time systems. The major results of the studies on the partition algorithm and the alternative algorithms are:

- The system substantially benefits from distributed scheduling in a wide range of laxi-

ties, group size, and communication cost. For example, for groups of size 6 and laxities between 1 and 3 seconds, using the partition algorithm, distributed scheduling improves the guarantee ratio by 15%.

- For most cases, the partition algorithm performs significantly better than the polling algorithm under different communication costs, different laxities, different types of precedence graphs (chains, trees, and general types), and different group sizes. For precedence chains, the polling algorithm performs better than the partition algorithm when the chain is long. When the group size is small, the performance of both the partition and polling algorithms is about the same. The guidelines for selecting between the partition algorithm and the polling algorithm under different laxities, different group size, and different types of precedence graphs are shown in Table 6.

- The sequential algorithm does not work well in distributed scheduling. For most of the tested cases, this algorithm performs only slightly better than the noncooperative algorithm.

- The system guarantee ratio increases as the laxity of groups increases. For example, for network load of 6 and groups of size 6, using the partition algorithm, the improvement is 20% as the laxity increases from 0.5 to 3 seconds.

Recall that in all these simulations, the cost of preprocessing a group, i.e., assigning scheduling windows to tasks was included in the scheduling overheads. As was pointed out in section 6.1.3, it is possible to compute these windows statically for different system states and dynamically use the appropriate window characteristics based on the existing state. This should decrease the scheduling overheads and hence further enhance the performance of the partition algorithm. In spite of the apparent complexity of the partition algorithm, it offers better performance than simple alternative algorithms over a wide range of system and application parameters.

The algorithm described here considers all task groups to be of equal criticalness. It is a straightforward extension to the presented algorithms to assign criticalness to task groups and to account for criticalness in performing the atomic guarantees.

References

[Che87] S. Cheng. *Dynamic Scheduling Algorithms for Distributed Hard Real-Time Systems.* PhD thesis, University of Massachusetts, Amherst, 1987.

[CSR87] S. Cheng, J. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. *IEEE Real Time Newsletter*, 3(2), 1987.

[GJ76] M.R. Garey and D. S. Johnson. Scheduling tasks with nonuniform deadlines on two processors. *J. ACM*, 23(3), 1976.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9, 1961.

Group size	Laxity		
	Small	Moderate	Large
Small	Partition	Partition/Polling	Partition/Polling
Moderate	Partition	Partition	Partition
Large	Polling	Polling	Partition

(a) Arbitrary precedence graphs

Group size	Tree	Chain
Small	Partition/Polling	Partition/Polling
Moderate	Partition	Partition/Polling
Large	Partition	Polling

(b) Precedence trees and chains (moderate and large laxities)

Table 6: Guidelines for selecting algorithms.

[KN85] H. Kasahara and S. Narita. Parallel processing of robot-arm control computation on a multiprocessor system. *IEEE Journal of Robotics and Automation*, 1(2), 1985.

[Law73] E. L. Lawler. Optimal scheduling of a single machine subject to precedence constraints. *Management Science*, 19, 1973.

[LY82] D. W. Leinbaugh and M. R. Yamini. Guaranteed response times in a distributed hard real-time environment. In *Proc. IEEE Real-Time Systems Symp.*, December 1982.

[MLT82] P.-Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Trans. on Computer*, C-31(1), 1982.

[RCG72] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. on Computer*, C-21(2), 1972.

[RS84] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1(3), 1984.

[SRC85] J. A. Stankovic, K. Ramamritham, and S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Trans. on Computer*, C-34(12), 1985.

[Ull76] J. D. Ullman. Complexity of sequence problems. In E. G. Coffman, editor, *Computer and Job-Shop Scheduling Theory*, J. Wiley, New York, 1976.