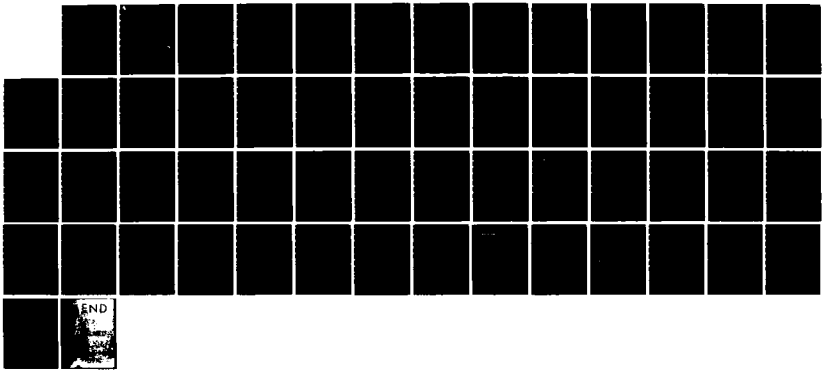
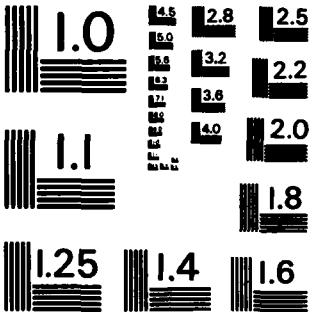
AD-A147 688 DECENTRALIZED CONTROL OF SCHEDULING IN DISTRIBUTED SYSTEMS(U) MASSACHUSETTS UNIV AMHERST DEPT OF ELECTRICAL AND COMPUTER ENGINEERING J A STANKOVIC 1/1

UNCLASSIFIED 14 JUN 84 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RESEARCH AND DEVELOPMENT TECHNICAL REPORT
CECOM

DECENTRALIZED CONTROL OF SCHEDULING IN DISTRIBUTED SYSTEMS

John A. Stankovic
Dept. of Electrical & Computer Engineering
University of Massachusetts
Amherst, MA 01003

Quarterly Report for the Period 15 MARCH 84 to 14 JUNE 84

Distribution Statement

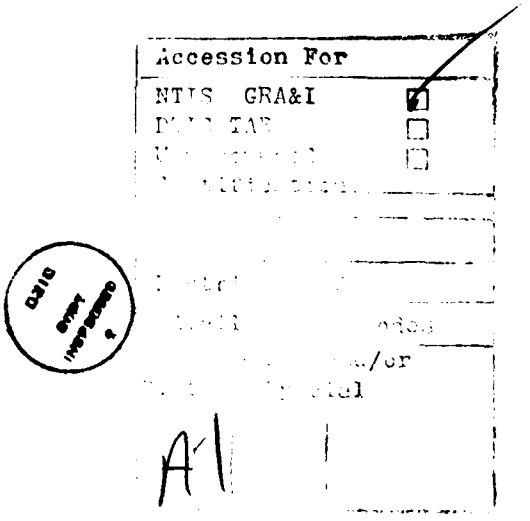
Approved for public release;
distribution unlimited

Prepared for
Center for Communications Systems (C. Graff)

CECOM
U S ARMY COMMUNICATIONS-ELECTRONICS COMMAND
FORT MONMOUTH, NEW JERSEY 07703

HISA-FM-1566

Table of Contents



1.0 Introduction

This report summarizes the work performed during the tenth quarter of this contract, from March 15, 1984 to June 14, 1984. The work in this quarter concentrated in three areas: simulations of our real-time bidding algorithm, non-traditional architectures studies for real-time systems, and scheduling tasks with resource requirements in hard real-time systems. Section 2 of this report describes the progress made in each of these areas.

Significant new results were obtained in the area of hard real-time constraints and a paper was written describing these results. A copy of the paper is attached as Appendix A.

2.1 Real-Time Constraints - Simulations

During the past few months, progress has been made in finalizing the simulation model of distributed hard real-time systems and evaluating the performance of a basic bidding algorithm in such a system. Significant results have been obtained which provides insight into the effectiveness of the algorithm. An improvement over the basic algorithm has been found. The improvement uses the prediction of future surplus time instead of the current surplus time in modifying bids. This report describes the basic algorithm and then describes the simulation results.

Each node in the distributed system has a scheduler. A set (possibly null) of guaranteed periodic tasks exist at each node. Aperiodic tasks may arrive at any node in the network. When a new task arrives, an attempt will be made to schedule the task at that node. If this is not possible, then the scheduler on the node interacts with the schedulers on other nodes, using a bidding scheme, in order to determine the node on which the task can be sent to be scheduled. The intent is to guarantee all periodic tasks and as many of the aperiodic tasks as possible, utilizing the resources of the entire network.

The scheduler is decomposed into two major system tasks: a local scheduler task and a bidder task. They may be invoked periodically or asynchronously. The local scheduler task invokes the guarantee routine to check to see if a newly arriving task can be guaranteed locally. If yes, the task is linked to the ready queue and is dispatched with an earliest deadline policy. If not, the task is linked to an outgoing RFB (request-for-bid) queue waiting to be processed by the bidder task. The bidder task has three phases. The first phase is to evaluate the tasks in the outgoing RFB queue and send out the RFB messages in behalf of the tasks. The second phase is to evaluate the incoming RFB message queue and send out the bids. The third phase is to evaluate the queue of returning bids to see if there is a node where the task can be sent to be scheduled.

In a hard real-time environment, the bidding algorithm must run in a timely fashion. Each RFB message is attached with a deadline of response. If the responding site cannot deliver the bid to the request site on time, no attempt will actually be made. Therefore, only viable bids will reach the request site and the communication overhead may be reduced. In reply to

a RFB message, a node estimated the surplus time available for the task between the estimated arrival time and the deadline of the task and sends the surplus time as the bid to the request site. The estimation is based on the assumption that the surplus time in the near future will be about the same as that in the recent past. This assumption has been found inadequate in a hard real-time environment and needs to be modified.

Our simulation has provided some insight into the performance of the bidding algorithm under various conditions (arrival rates, processing power of system tasks, length of memory accumulation periods). Several baseline algorithms are simulated to bound the performance of the bidding algorithm. The performance matric used is the number of tasks guaranteed in the network-wide system. Other statistics collected include the utilization factors and the turnaround time. The turnaround time is defined to be the time interval between the arrival time and the time when a task is guaranteed or found to be impossible to guarantee (locally or at remote sites).

Table 1 illustrates the number of tasks guaranteed under various arrival rates of aperiodic tasks. The percentage of the total number of tasks guaranteed decreases and the turnaround time at heavily loaded sites increases as the arrival rate is increased. This is because the processing power of the system tasks is limited at the heavily loaded sites and only a certain number of messages may be processed in time. Also shown in the table, at rate 3 and 4, the input stream of aperiodic tasks starts to saturate the processing power of the system tasks to the load factor of the aperiodic tasks.

	rate 1	rate 2	rate 3	rate 4
Percentage of Tasks Guaranteed	97.6%	89.3%	77.5%	63.7%
Normalized Total No. Tasks Guaranteed	1	1.023	0.966	0.849
Turnaround Time				
host 1	1.5 msec	1.1 msec	1.1 msec	1.0 msec
host 2	362.0	608.4	916.1	3364.6
host 3	1.0	1.0	1.0	1.0
host 4	4.3	43.3	96.4	121.8
host 5	355.0	662.0	854.4	1112.9
Utilization Factor				
host 1	0.69	0.76	0.68	0.50
host 2	1.00	1.00	1.00	1.00
host 3	0.66	0.72	0.66	0.45
host 4	0.99	1.00	1.00	1.00
host 5	1.00	1.00	1.00	1.00

Table 1. Performance of the Basic Bidding Algorithm under Different Arrival Rates of Aperiodic Tasks.

We have also evaluated the effect of changing the seeds of the random number generators on the statistics. Four set of seeds have been used. They only result in 2.6% difference in the total number of guaranteed tasks. This convinces is that 10% improvement over the performance matrics is significant.

The effect of varying the length of the memory accumulation period (LMAP) is also evaluated. As shown in Figure 1, for three arrival rates the LMAP is varied from 500 msec to 3,000 msec. Only a small perturbation in the percentage of tasks guaranteed (less than 1.9%) is observed. This insensitivity to LMAP is further investigated by analyzing the variation of the surplus time during the run time.

As shown in Figure 2, the surplus times are recorded and plotted as a function of the simulation time. Several interesting things are observed. First, less transient behavior is exhibited by the curves in which higher LMAP's are used. Second, there is large sinusoidal pattern of surplus times at lightly loaded hosts (host 1 and 3). Third, the variations of the surplus time factor for host 1 and 3 complement each other.

Since the surplus at host 1 and 3 vary alternatively, they also take turns in dominating in the bidding process. When the surplus for one host is increased above the cross over point, the host starts to take over the bidding, while the other ceases to bid. But as long as the dominating host has enough surplus, viable bids will be issued and task will be awarded to this site. Therefore, performance depends in whether there is a host with high surplus most of the time. When all the hosts are low in surplus, few viable bids are issued resulting in few guaranteed tasks due to bidding.

From the sinusoidal pattern of surplus time, we infer there is an overbidding and an underbidding effect in the peak area and the valley area of the curves, respectively. In the peak area, because of the high current surplus and the unawareness of the outcome of the bidding, the number of bids issued may be more than that can be accomodated by the host. As a result, conflicts are more likely to occur among task with the same hard real-time constraints. In the valley area, because of the low current surplus and the unawareness of the shrinking set of local guaranteed tasks, the host may withdraw itself from bidding and stay idle too long until it finds out that it has high surplus again. Because of the delay, the host will waste cpu time in waiting for tasks to be awarded. The underbidding effect will result in the low utilization of the system.

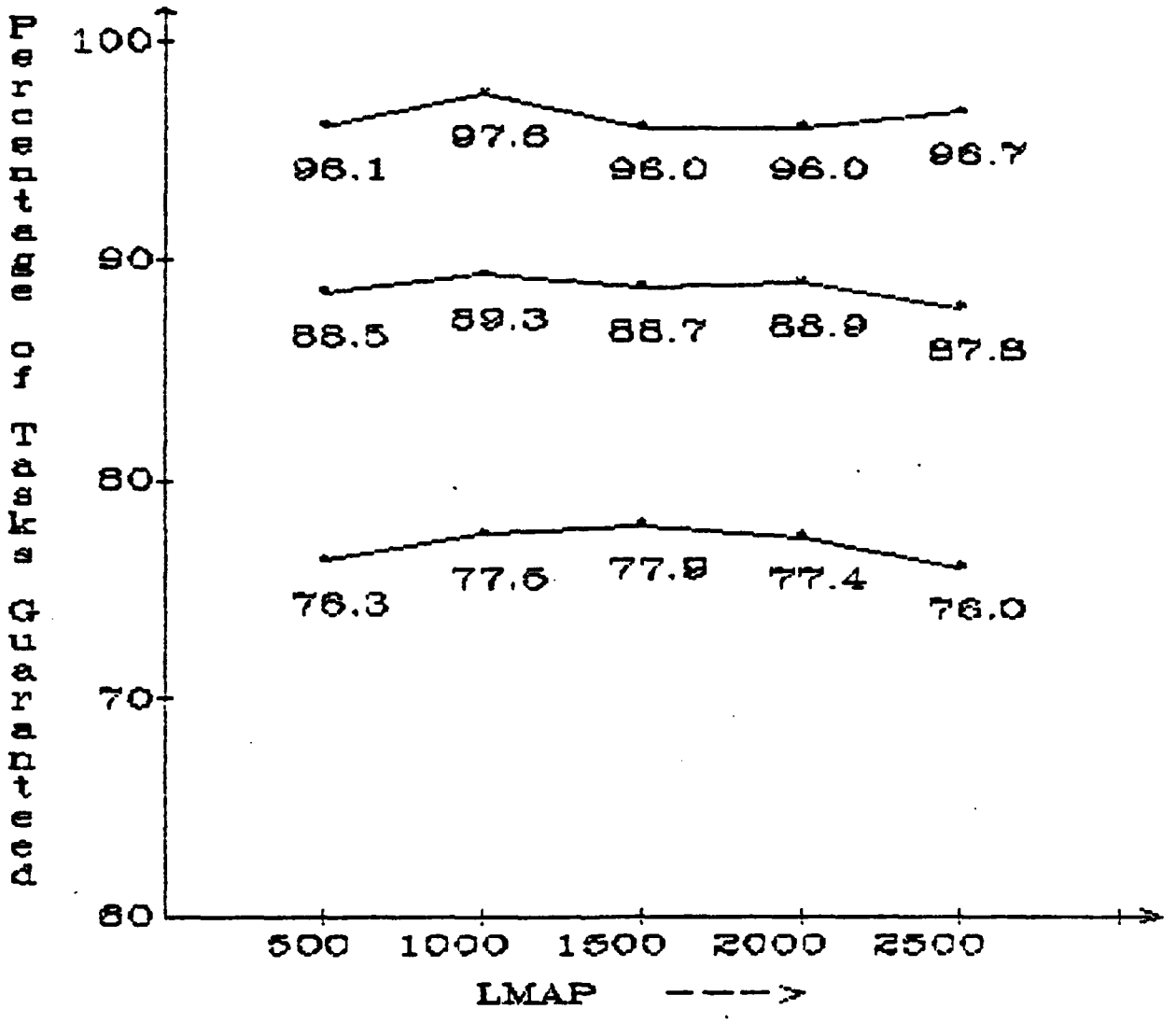
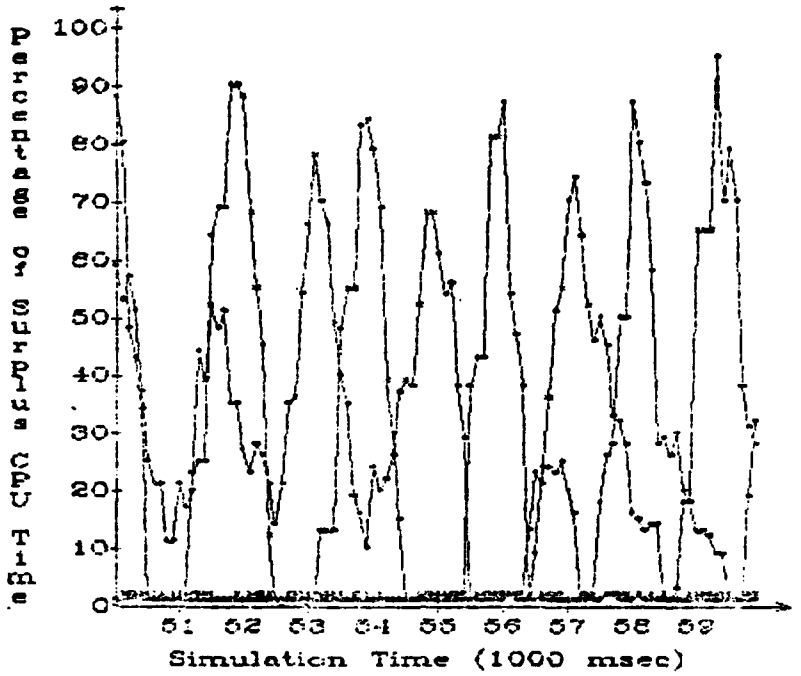
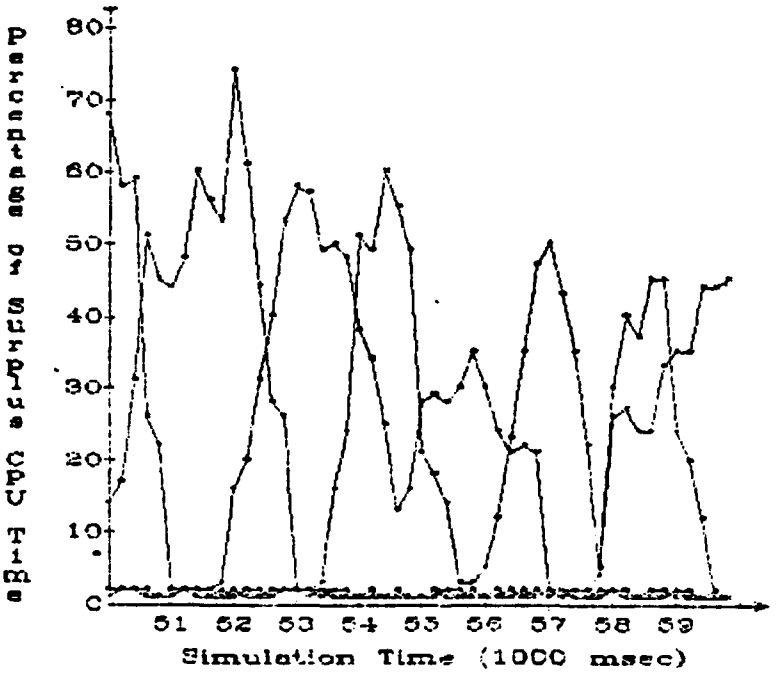


Figure 1. Effect of varying LMAP.

Figure 2

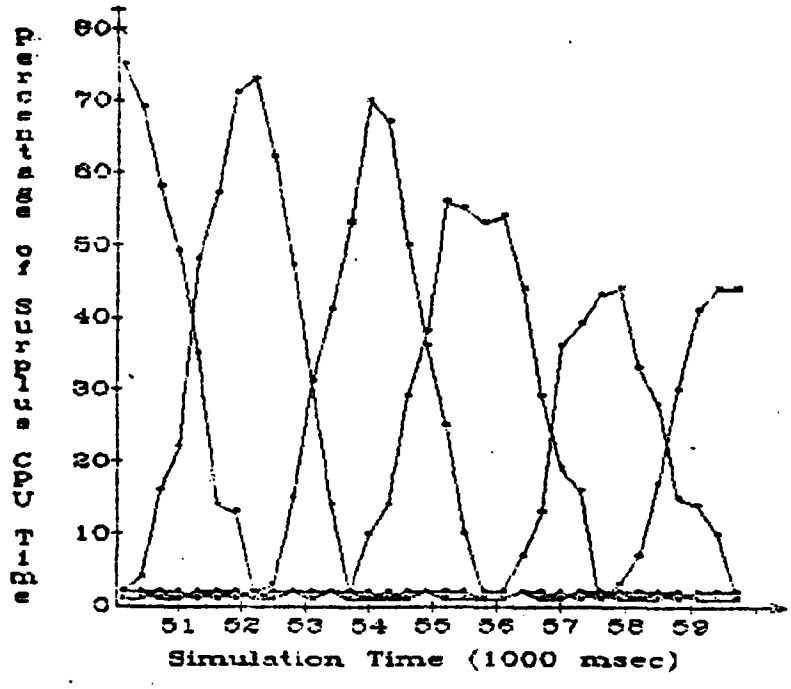


(a) LMAP = 500 msec.

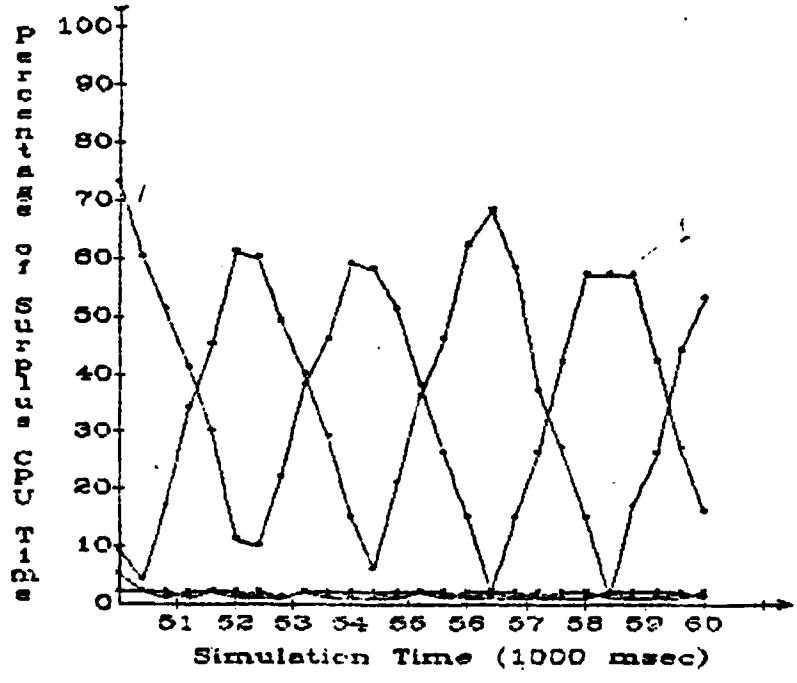


(b) LMAP = 1000 msec.

Figure 2 (Continued)

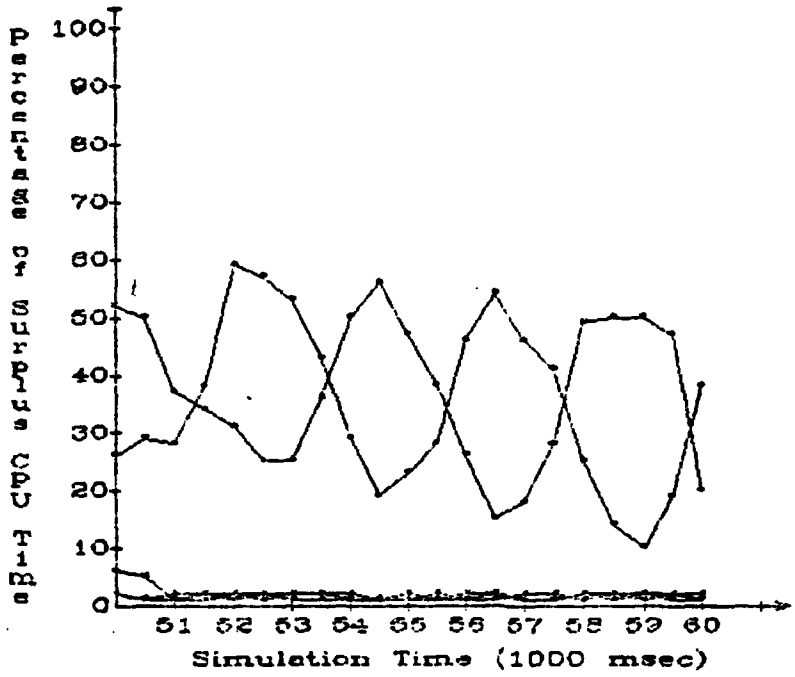


(c) LMAP = 1500 msec.

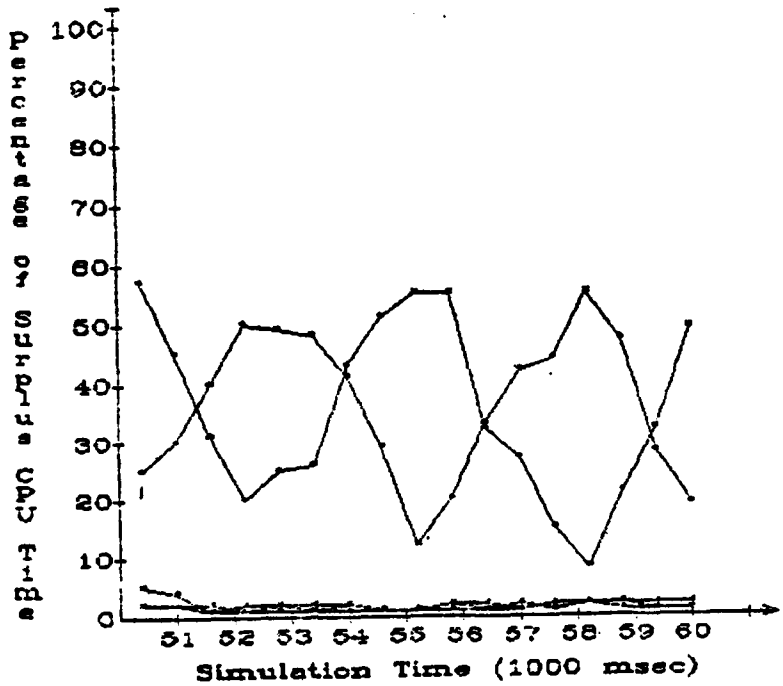


(d) LMAP = 2000 msec.

Figure 2 (Continued)



(e) LMAP = 2500 msec.



(f) LMAP = 3000 msec.

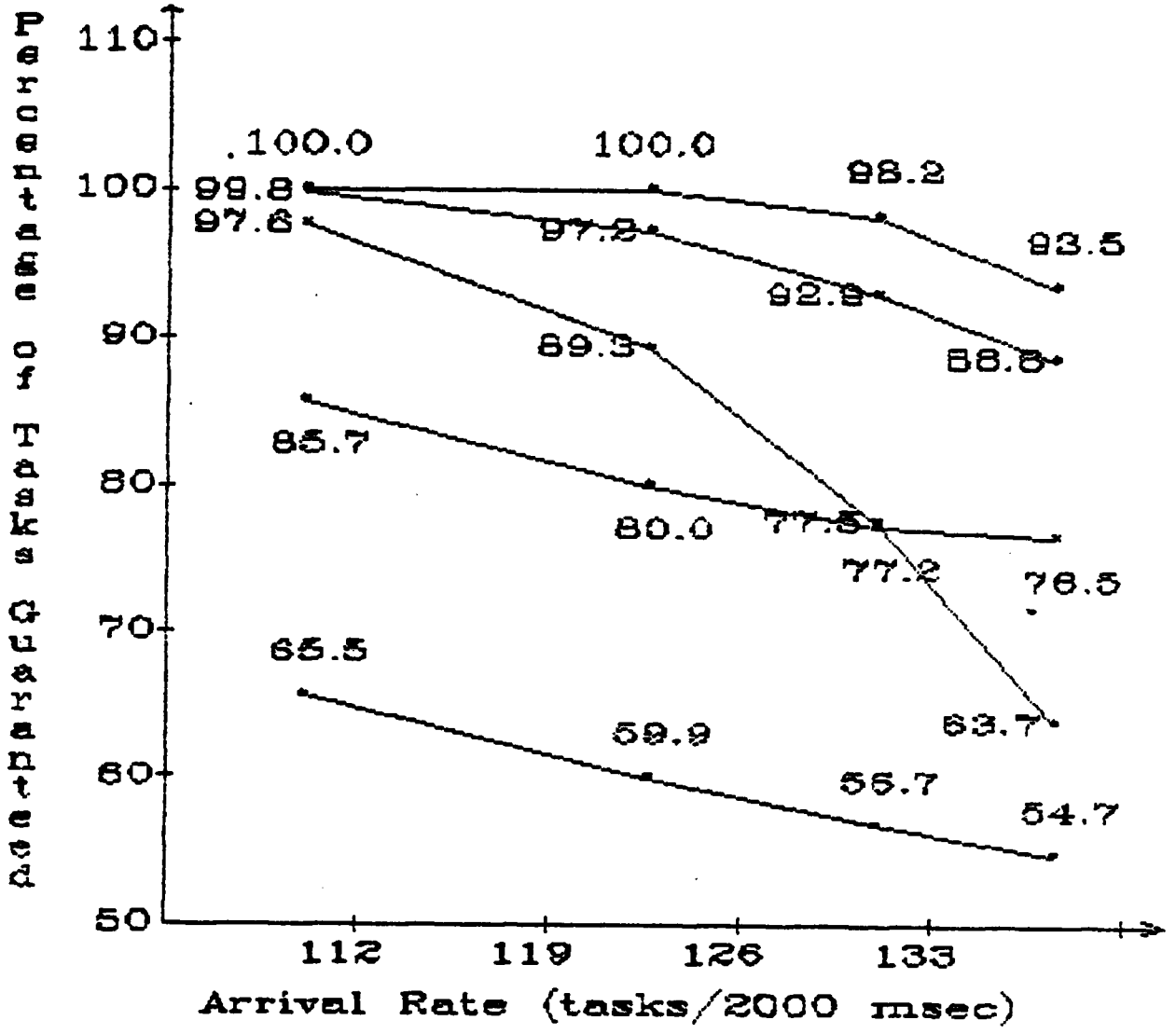


Figure 3. Bidding and baseline algorithms.

From the overbidding and underbidding effects, we conclude that the original algorithm which based the estimation of surplus in the near future on the surplus in the recent past is not adequate for the bidding algorithm in distributed hard real-time systems. The estimation should be based in the prediction of the future task load instead. As a result, the bids issued by the hosts are more accurate which hopefully will result in more uniform utilization of the system. A proposed approach is described as follows: Let FTIME be the maximum free time available in each LCM of the periods of the periodic tasks, i.e., FTIME is equal to LCM minus CPU time reserved for the periodic tasks. The surplus time available for a task of which the estimated arrival time and the deadline are ART and D, respectively, is equal to

$$\frac{D-ART}{LCM} * FTIME - \text{CPU time needed for future tasks,}$$

where the future tasks include guaranteed tasks of which the deadlines are between ART and D, and the tasks which will be arriving and guaranteed as a result of the bidding. The estimation of the number of tasks which will be arriving and guaranteed may be a heuristic function of the number of outstanding bids and the surplus time ratio between local host and other hosts. We expect that this approach will eliminate the sinusoidal pattern and achieve higher utilization of the system resources. This new approach is currently under investigation.

We have also made a simulation runs to bound the performance of the bidding algorithm. Several baseline algorithms were conceived: 1) Perfect state information algorithm (PSA) - Each node is assumed to have perfect state information about every other nodes so that the local schedular is capable of determining whether a task can be guaranteed at any other site or not. 2) Current utilization information algorithm (CUA) - Each node is assumed to have up-to-date CPU utilization information about every other nodes all the time, and the decision of choosing the node to send the task is based on utilization. 3) Random scheduling algorithm (RSA) - Each node randomly chooses a node whenever a task cannot be guaranteed locally. 4) Non-cooperative algorithm (NCA) - No cooperation among nodes is assumed. If a task cannot be guaranteed locally, it simply fails. PSA and CUA impose upper bounds on the percentage of tasks guaranteed, while RSA and NCA impose lower bounds.

The simulation are run for various arrival rates of aperiodic tasks. At rate 1 and 2, the numbers of tasks arriving in each least common multiple of periods of periodic tasks are 84.5% and 94% of the maximum number of tasks guaranteeable, respectively. As shown in Figure 3, the bidding algorithm outperforms the lower bound algorithms significantly at rate 1 and 2, which are pretty high arrival rates of tasks.

From the simulation results described above, we conclude that the bidding algorithm is effective in scheduling tasks in the distributed hard real-time systems and even with a basic bidding algorithm, it yields satisfactory performance under heavy loads. The improvement needed for the basic

bidding algorithm is to use the prediction of the arriving tasks in the near future in the estimation of the future surplus time so that the overbidding and the underbidding effects may be eliminated. Under saturation condition, the performance degrades which should be avoided by adjusting the processing power of the system tasks if there is surplus time available.

2.2 Real-Time Constraints - Non-traditional Architectures

Under this contract we have developed a flexible scheduling algorithm for distributed real-time systems. This approach is based on a local guarantee algorithm and a network-wide bidding algorithm. A task that enters a host is scheduled to run on it if it can be guaranteed to execute by its deadline. If not, the local scheduler calls a bidder routine which interacts with the other hosts to determine if the task can be elsewhere. Thus the other hosts "bid" for the task in response to the "request for bid" initiated by the local host.

In this scheme, every host posses a group of interacting processes: the local guarantee routine, the bidder, the scheduler and user tasks. Because of the logical separation of these processes, we feel that the hosts can be designed as multiple processor systems, with each of the above processes mappings onto separate processors. The two main advantages of this scheme are those of increased performance and fault tolerance. Performance improvements are possible because the CPU is relieved of the overhead of the bidder, scheduler etc. Because of the proliferation of processors at each node, it is now also feasible to provide some degree of fault tolerance.

Preliminary studies indicate that a feasible host architecture might incorporate multiple microprocessors interacting through some communications medium. We are currently studying various architecture that seem appropriate. These studies include queueing models as well as simulation systems of these architectures. Because our algorithm might generate a large number of messages in the subnets, we are also planning to study various network architectures which provide good connectivity and also reduce the number of messages in the subnet.

Many real-time systems also have another requirement, i.e., robustness. We are beginning to investigate the possibilities of fault tolerance in our system at two level:

1. Because of the availability of more than one processor at the host, we feel that it is possible to incorporate tolerance to failures at a processor by dynamic allocation of its tasks among the fault-free ones. The key factor here is how to decide if a processor is faulty. We are investigating alternative schemes to do this.

2. At the network level, we would like to be able to switch tasks between hosts in the event of a total failure at a host.

2.3 Real-Time Constraints - Resource Requirements

We have developed a heuristic approach for solving the problem of scheduling tasks with deadlines and general resource requirements. This is an extension of the basic algorithm we developed, and whose simulation results were presented in Section 2.1. The crux of our approach lies in the heuristic function we developed. This function weighs three factors. These factors quantify a task's requirements (deadlines, CPU execution-time requirement and resource requirements) in such a way as to help develop a feasible schedule quickly. If a feasible schedule is found then the set of tasks are guaranteeable. We have done extensive simulation to show that our approach works. We believe that the results we have developed in this area are significant. The results have been fully described in a paper written during this quarter and attached as Appendix A.

3.0 Conclusions and Recommendations

In this quarter we have concentrated in scheduling tasks with hard real-time constraints and made significant progress. We will continue to aggressively pursue all research areas described in this report. Further, we will begin to spend more time on scheduling tasks with soft real-time constraints. The immediate problems in this area are a full understanding and modeling of clusters of processes and distributed groups of processes.

4.0 Budget

Spent prior to the months listed below $66,464.62.

	Planned	Actually Spent
December (83)	$4000.00	$3319.21
January (84)	4000.00	2336.68
February	4000.00	7323.58
March	4000.00	5376.75
April	4000.00	#513.36
May	4000.00	
June	7000.00	
July	7000.00	
August	7000.00	
September	5000.00	
October	5000.00	
November	5000.00	
December	4365.38	

Total: Planned and Spent Prior to Months Listed Above. $130,830.00 Total Spent: $17,842.86

Modified Budget

	Planned
May (84)	$4000.00
June	8000.00
July	8000.00
August	7000.00
September	5000.00
October	5000.00
November	5000.00
December	4522.52
Total Planned	$46,522.52
Total Spent	84,307.48
Total	$130,830.00

5.0 Appendix A

SCHEDULING TASKS WITH RESOURCE REQUIREMENTS IN HARD REAL-TIME SYSTEMS[1]

Wei Zhao

Krithivasan Ramamritham

Computer and Information Science

John A. Stankovic

Electrical and Computer Engineering

University of Massachusetts

Amherst MA 01003

May 1984

1. This work was supported in part by the US Army CECOM, CENCOM under grant DAABO7-82-K-JO15 and by the NSF under grant MCS 82-02586.

SCHEDULING TASKS WITH RESOURCE REQUIREMENTS IN HARD REAL-TIME SYSTEMS

ABSTRACT

This paper describes a heuristic approach for solving the problem of scheduling tasks with deadlines and general resource requirements in a dynamic real-time system. The crux of our approach lies in the heuristic function used to select the task to be scheduled next. The heuristic function is weighted three factors. Each factor explicitly considers information about real-time constraints of tasks and utilization of resources. Simulation studies show that the weights for the various factors in the heuristic function have to be fine-tuned in order to obtain a degree of success comparable to that obtained via exhaustive search. In addition, modifying the heuristic to use limited backtracking improves the degree of success substantially. This improvement is observed even when the initial set of weights are not tailored for a particular set of tasks. Simulation studies also show that in most cases the schedule determined by the heuristic algorithm is optimal or close to optimal.

1. Introduction

Loosely coupled, real-time distributed systems are becoming more prevalent in applications such as nuclear power plants and process control [SCHO84]. These systems contain many tasks which have severe real-time constraints. In fact, these applications require that these tasks have execution deadlines that must be met and are thus said to have *hard real-time* constraints. In practice, many solutions used today assume complete and prior knowledge of the task sets, including their deadlines, worst case computation times, precedence relations and resource requirements. Incorporating all this knowledge in a static way makes the system inflexible and costly (each system is unique). Also, most current research in this area is restricted to multiprocessing systems, is compute bound and does not adapt to the dynamics of the systems. Our goal is to develop a flexible scheduling algorithm for loosely coupled distributed systems that does not require complete and prior knowledge, is not compute bound, and quickly adapts to the dynamics of the "entire" system. Our basic algorithm and approach have been reported in [RAMA84]. This paper reports on a sophisticated extension to our basic algorithm. In the remainder of this section, we briefly describe our basic approach and identify exactly how this paper significantly extends the basic algorithm.

Our basic approach [RAMA84] assumes that each node of a loosely coupled distributed system contains a local scheduler, a bidder and a dispatcher. The *local scheduler* at a node is invoked when a new task arrives at that node. It decides if the new task can be *guaranteed* at this node. The guarantee means that no matter what happens (except failures) this task will execute by its deadline, and that all previously guaranteed tasks will also still meet their deadlines. If the new task cannot be guaranteed locally, then the new task is either terminated (without executing) or is sent to another node chosen on

the basis of a *bidding scheme*. The *bidder* on a node is responsible for 1) sending out requests for bids for a task that cannot be guaranteed locally, 2) evaluating bids from other nodes, and 3) sending the task to the best bidder. It also makes bids in response to requests from other nodes. The *dispatcher* is the system program that actually schedules the guaranteed tasks. It should be pointed out that when a node bids for a task, it does not reserve CPU time for that task. Reserving CPU time ties up too many resources for too long a time. Consequently, when a bidding task finally arrives at a bidder node, the node will again try to guarantee it. In case this guarantee fails, the task may be re-bidded again, or declared as not "guaranteeable". In our approach, each of the functions of the local scheduler, bidder, and dispatcher is carefully controlled in order to handle real-time constraints.

In our basic algorithm [RAMA84], a task is characterized by a computation time and a deadline. Tasks are assumed to be independent. We also assume that resources are always available to the executing task. In this paper, we relax this assumption. We now allow a task to request any number of resources, including CPU, I/O devices, files, etc. This means that our local scheduling algorithm must be modified to consider not only computation time and deadlines of tasks but also their resource requirements. This is a significant extension to the previous scheduling algorithm. This paper focuses on the local scheduler portion of the scheme. In particular, we describe a technique for deciding whether a set of tasks on a node can be guaranteed to execute on that node.

The work described in this paper is not only an extension to our own work, but also to work in the field in general. Most related work on scheduling tasks with hard real-time constraints is restricted to uniprocessor and multiprocessor systems [MUNT70, LIU73, DERT74, JOHN74, GARE75, MOK78, and TEIX78] and typically do not take tasks'

resource requirements into account. In his work, Leinbaugh [LEIN80 and LEIN82] dealt with resource requirements. He developed analysis algorithms which, when given the resource requirements of each task, determine an upper bound on the response time of each task. While his approach is useful at system design time to statically determine the upper bounds on response times, there is no attempt at guaranteeing that a new task will meet its deadline.

On the other hand, because of the difficulty of the problem, heuristic approaches have been taken for related scheduling problems [EFE82 and MA84], that is, heuristics are used in their systems. According to Lenat [LEN82 and LEN83], heuristics are informal, judgmental rules of thumb which come in two types: those that *actively* guide the system toward plausible paths to follow and those that guide the system away from implausible ones. Ma and Efe use heuristics which guide their systems away from implausible paths. This approach of only using the second type of heuristic is limited because in the worst case, the exponential problem cannot be avoided [EFE82 and MA84]. In our heuristic algorithm, we use both types of heuristics. We develop a heuristic function which synthesizes the various factors of real-time scheduling considerations to *actively* direct the scheduling process to a plausible path. We also constraint the search space by looking only at *strongly feasible* paths, preventing us from looking at implausible paths. As a result, even in the worst case, our scheduling algorithm is not exponential. Moreover, simulation results show that our heuristic algorithm can guarantee an arbitrary set of feasible tasks almost all the time. It is also shown that in most cases the schedules found by our heuristic scheduling algorithm are optimal or close to optimal.

In the remainder of this paper, Section 2 provides an overview of our heuristic approach for incorporating resource requirements into dynamic hard real-time scheduling while Section 3 presents the algorithm for the local scheduler, and discusses the heuristic function in detail. An evaluation of the algorithm, in Section 4, shows that the algorithm is effective. However, this initial evaluation also suggests improvements that are presented and evaluated in Section 5. Analyses of the complexity and optimality of the algorithm are made in Section 6. Section 7 concludes the paper with discussions of the achievements and extensions of this study.

2. Strategy and Function of the Scheduler

This section first defines and discusses the basic terms needed to understand our scheduling algorithm. It then outlines the strategy followed by the scheduler, and finally discusses an analogy between scheduling and searching.

2.1 Resources and Tasks

Let each *node* of a loosely coupled distributed system contain a set of resources, R_1, R_2, ..., R_r. A resource can be serially shared by tasks. The types of resources include CPU, physical devices, and files. A resource is *active* if it has processing power, otherwise, it is *passive*. For example, a CPU or a physical device is active, but a file is passive. Thus, a passive resource must be used with some active resource.

A *task* is a scheduling entity. The following characteristics of a task, T, are assumed known when it arrives:

1. The worst case computation time, C(T);
2. The deadline, D(T), by which the task must complete;
3. The resource requirements of the task. It is assumed that a task needs its resources throughout its execution.

A task will request at least one active resource and zero or more passive resources.

A task is said to be *guaranteed*, if the scheduler can decide on a start time, so that all the resources needed by the task are available by that time, and the task will meet its deadline. A *feasible schedule* is a list of tasks that have been guaranteed. With respect to a set of tasks, a schedule is *full*, if it contains all tasks in the set, otherwise it is *partial*. A schedule $(T_1, T_2, ..., T_s, T_{s+1})$ is an *immediate extension* of the schedule $(T_1, T_2, ..., T_s)$.

2.2 Strategy of the Scheduler

The basic strategy we employ to determine whether or not a new task can be guaranteed on a node is as follows:

1. On each node, there is always a set of guaranteed tasks, say n, $n \geq 0$, and a current full feasible schedule for these tasks.

2. When a new task arrives, it can be guaranteed if a new full feasible schedule exists for the n+1 tasks. That is, the n tasks in the current feasible schedule remain guaranteed and the deadline of the new task also can be met.

3. If the new task is guaranteed, then the new feasible schedule, containing the new task and the n old tasks, replaces the current one. This feasible schedule determines all the start times for the tasks currently on the node, and will not be altered until another new task is guaranteed.

4. If the new task cannot be guaranteed, that is, the local scheduler cannot find a full feasible schedule for all n+1 tasks, bidding is invoked [RAMA84]. The new task is sent to the node offering the best bid. The original current feasible schedule for the tasks previously guaranteed on the sending node remains unchanged.

In the discussion of our heuristic in this paper, we assume that the current running task is preemptable on all its resources. A minor modification is required to deal with the situation where the currently running task is not preemptable. We discuss this modification in Section 3.1.

In the rest of this paper, we will focus on the algorithm for determining whether a full feasible schedule exists for a given set of tasks.

2.3 Scheduling and Searching

The scheduler determines a full feasible schedule for a given set of tasks in the following way: It begins with an empty schedule and tries to extend it with one task at a time until a full feasible schedule is derived. This is, in fact, a search problem. The structure of the search space is a *search tree*. The *root* of the search tree is the empty schedule. An *intermediate vertex* of the search tree is a partial schedule. A *descendant* of a vertex is an immediate extension of the schedule corresponding to the vertex. A *leaf*, a terminal vertex, is a full schedule. Note that not all leaves will correspond to feasible schedules. The goal of our algorithm is to search for a leaf that corresponds to a full feasible schedule.

3. The Scheduling Algorithm

This section describes the heuristic scheduling algorithm. We first present several data structures used, motivate a constraint on the search process, and then present the algorithm. An extension to the algorithm is discussed in Section 5.

3.1 Data Structures

The algorithm maintains a vector EAT, to indicate the *Earliest Available Times* of resources:

$$EAT = (EAT_1, EAT_2, ..., EAT_r)$$

where EAT_i is the earliest time when resource R_i will become available. Initial values of EAT_i for all i will be 0 if the running task is preemptable. Otherwise, EAT_i will be the time when the running task finishes using it. Each time the partial schedule is extended, EAT will be updated taking into account the newly added task's resource requirements and completion time.

At each level of the search tree, the scheduler computes EST(T) and New-EAT(T) for each task T that remains to be scheduled. EST(T) indicates the *Earliest Start Time* of task T if it is scheduled next. Since a task T can run only when all resources it needs are available, we define EST(T) as

$$EST(T) = MAX(EAT_i \text{ where } T \text{ needs } R_i).$$

New-EAT(T) is a vector with the same size as EAT and contains the EAT values if task T is scheduled next. In other words, New-EAT(T) will replace the current EAT if task T is scheduled. The computation of New-EAT(T) is illustrated by the following example.

EXAMPLE Assume we have 7 resources R_1, R_2, ..., R_7, and among them R_1, R_2, R_3, and R_4 are active resources. Let current EAT be

(1) $EAT = (EAT_1, EAT_2, EAT_3, EAT_4, EAT_5, EAT_6, EAT_7)$

$= (5, \quad 10, \quad 25, \quad 15, \quad 10, \quad 15, \quad 5)$, and

let task T with computation time C = 10, and with resource requirements R_1, R_2 and R_5, be one of the tasks that remain to be scheduled.

First, the EST, the Earliest Start Time for T is

(2) $EST(T) = MAX(EAT_1, EAT_2, EAT_5)$

$= MAX(5, \quad 10, \quad 10) = 10.$

Then New-EAT(T) should be

(3) New-EAT(T) = (20, 20, 25, 15, 20, 15, 5)

because T is assumed to start at its EST and it uses 10 units of computation time while holding R_1, R_2, and R_5. R_1 will be idle from time 5 to 10. Our scheduling algorithm is designed to minimize such idle times.

New-EAT(T) may be further refined if we distinguish active resources from passive ones. Because a passive resource must be used with active ones, no task can use them until

time = MIN(New-EAT_i where R_i is an active resource)

= MIN(New-EAT_1, New-EAT_2, New-EAT_3, New-EAT_4) = 15.

Thus, (3) should be further updated as

(4) New-EAT(T) = (20, 20, 25, 15, 20, 15, 15)

That is, all New-EAT$_i$s for passive resources should not be less than (therefore must be updated to be equal to) the minimum New-EAT$_i$ for active resources.

At?M each level of the search, the scheduler also calculates a vector called DRUR, the *Dynamic Resource Utilization Ratio*, which indicates the degree to which tasks that remain to be scheduled will use resources:

$$DRUR = (DRUR_1, DRUR_2, ..., DRUR_r)$$

where $DRUR_i$ is defined as

$$DRUR_i = \frac{\Sigma\ (\ C(T),\ T \text{ remains to be scheduled and uses } R_i)}{MAX(D(T),\ T \text{ remains to be scheduled and uses } R_i) - EAT_i}$$

where i = 1, ..., r.

Note that all $DRUR_i$ of a DRUR associated with a partial feasible schedule should be less than or equal to 1 for the remaining tasks to be scheduled. EAT, New-EATs and DRUR are updated each time the partial schedule is extended.

3.2 A Constraint on the Search

Using the data structures, EAT and DRUR, described above, we can state a constraint on the search for a full feasible schedule.

We define a feasible partial schedule to be *strongly feasible* if

1. DRUR associated with the schedule has $DRUR_i \leq 1$ for i = 1, ..., r; and
2. all of its immediate extensions are feasible.

A full feasible schedule is defined to be strongly feasible.

If a schedule is not strongly feasible because condition 1 or 2 fails, then the failed condition will also fail for all descendants of the non-strongly feasible schedule. Hence, none of the descendants of a non-strongly feasible schedule can be strongly feasible. On the other hand, the ancestor of a full feasible schedule must be strongly feasible, otherwise the full schedule itself will not be feasible. Therefore, only strongly feasible schedules can lead to a full feasible schedule. Considering this fact, a constraint on the search for a full feasible schedule is :

For a partial schedule to be extensible to a full feasible schedule, the partial schedule should be strongly feasible.

From the viewpoint of the algorithm, this means that it is not necessary to search through a vertex corresponding to a non-strongly feasible schedule, because a non-strongly feasible schedule will not lead to a full feasible schedule.

By the above constraint, the search is confined only to those vertices with strongly feasible schedules. However, in the worst case an exhaustive search may still be required, which is computationally intractable. In order to make the algorithm computationally tractable, even in the worst case, we choose only one of the vertices at each level to expand the search tree. The vertex chosen is the one which appears to be most capable of leading to a full feasible schedule. It is in this choice that our algorithm provides a way of actively guiding the search toward a plausible path — a feature that distinguishes our

approach from those of others. In the next subsection, we discuss our algorithm that incorporates the heuristics necessary to make this choice.

3.3 The Scheduling Algorithm

The pseudo code for our heuristic scheduling algorithm is given in Fig 1. Beginning with the empty schedule, the algorithm searches the next level by expanding the current vertex (a partial strongly feasible schedule) to only one of its immediate descendants. If the immediate descendant is also a strongly feasible schedule, the search continues until a full feasible schedule is met. At this point, the searching process (i.e., the scheduling process) succeeds and all the tasks are known to be guaranteed. If at any level, a non-strongly feasible schedule is met, the algorithm announces that the searching (scheduling) process fails and that not all the tasks can be guaranteed. It is possible to extend the algorithm to continue the search even after a failure is found, and this extension is discussed in Section 5.

```
PROCEDURE Scheduler(task-set : task-type; VAR schedule : schedule-type);

("Parameter task-set is the given set of tasks to be schedule ")
("Function H is the heuristic function, being discussed in 3.4")

BEGIN
  schedule := empty;
  WHILE NOT empty(task-set) DO
    BEGIN
      calculate EST and New-EAT for each task in task-set;
      calculate DRUR;
      IF not strongly-feasible(schedule) THEN
        return(not guaranteed)
        ELSE
        BEGIN
          apply function H to each task in the task-set;
          let T be the task with the minimum value of function H;
          task-set := task-set - { T };
          schedule := schedule + { T };
          EAT := New-EAT(T);
        END;
    END;

  return(guaranteed);
END;
```

Fig 1 : The Heuristic Scheduling Algorithm

Clearly, at each level of the search, effectively and correctly identifying the immediate descendant is important for the success of the algorithm. This is the core of the algorithm. We construct a heuristic function named H to help in the selection process. At each level, function H is applied to all the tasks remaining to be scheduled. The task with the minimum value returned by function H is selected to extend the schedule. The next subsection discusses function H in detail.

3.4 The Heuristic Function

Function H is defined as follows:

$$H(T) = W_1 * X_1(T) + W_2 * X_2(T) + W_3 * X_3(T)$$

where T is a task, W_1, W_2 and W_3 are weights and

1. X_1 takes into account the resource requirements of T as well as the resource utilizations. We will further develop X_1 later.

2. X_2 equals the laxity of task T, and hence is given by

$$X_2(T) = D(T) - (EST(T) + C(T))$$

where EST(T) is the Estimated Start Time for task T according to the availabilities of resources it will request; and C(T) is the computation time of T.

3. $X_3 = C(T)$.

Note that, if W_1 and W_2 are zero and $W_3 = 1$, then the task with the least computation time C will be selected first; if $W_2 = 1$ but W_1 and W_3 are zero, the task with the least laxity will be selected first. Simulation results given in Section 4 show that neither of the scheduling polices are appropriate. However if the three weights are properly chosen, we find that function H provides effective selection information.

Now, we further elaborate the first part of function H. X_1 is defined as:

$$X_1 = DRUR \bullet DRIF$$

where $\bullet$ stands for *dot product*; DRUR is the Dynamic Resource Utilization Ratio associated with the current partial schedule, as defined in section 3.1; DRIF is the *Dynamic Resource Idle Factor*.

The Dynamic Resource Idle Factor, DRIF, is a function of the task being considered for expanding the current feasible schedule. It indicates the amount of time each resource would remain idle if the task is selected next.

Before formally defining DRIF, we discuss three different types of idleness. Consider the example in 3.1 again: In that example, EAT = (5, 10, 25, 15, 10, 15, 5); a task T remaining to be scheduled has C(T) = 10 and needs R_1, R_2, and R_5. Therefore, EST(T) = 10 and New-EAT(T) = (20, 20, 25, 15, 20, 15, 15). We use a graph shown in Fig 2 to indicate EAT, New-EAT(T) as well as different idle times:

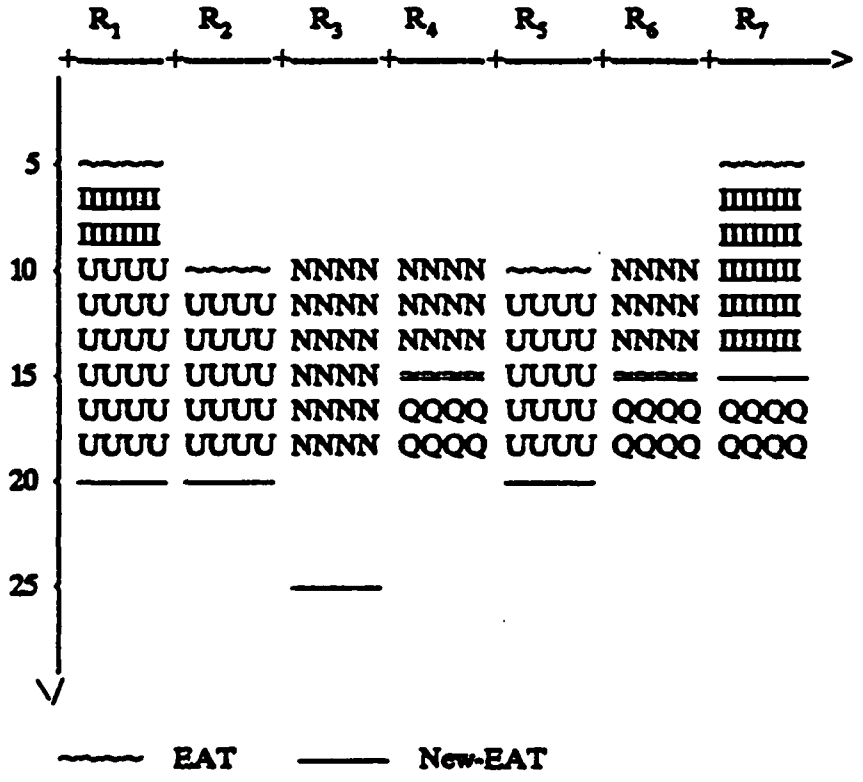


Fig 2 : EAT, New-EAT(T) and Idle Times

Task T uses R_1, R_2, and R_5 in the area marked "U". The other three areas marked "T", "Q" and "N" indicate three different types of idle time.

1. The area marked "T" is the idle time of R_1 and R_7. No task can use this time in the future if T is scheduled next. Therefore this indicates time that is *wasted* if T is scheduled next.

2. The area marked "N" is the time task T may run in parallel with some other task. For example, since R_3 becomes available only at time 25 then some task may be using R_3 until that time. Hence by scheduling T next, there is possible parallelism between T and the task using R_3. That is, the area indicated by "N" is, in some sense, *negative* idle time.

3. The idle time indicated by "Q" could be used or wasted, depending on the tasks selected later. For example, if the next task needs R_4 and R_6, then the time areas marked by "Q" under R_4 and R_6 will be used. However if the next task needs R_1, R_4 and R_6 then the areas marked by "Q" will be wasted idle time.

By the above example, it is clear that the three different idle times should be taken into account differently because they have a different impact on the system. We define

$$\text{DRIF}(T) = \text{DRIF-I}(T) + \text{DRIF-N}(T) + \text{DRIF-Q}(T)$$

where DRIF-I, DRIF-N, and DRIF-Q represent the three types of idle times on resources.

DRIF-I corresponds to the "T" idle time on each resource, and is defined as

$$\text{DRIF-I}(T) = \begin{Bmatrix} \text{DRIF-I}_1(T) \\ \text{DRIF-I}_2(T) \\ \vdots \\ \vdots \\ \text{DRIF-I}_r(T) \end{Bmatrix}$$

where for i = 1, ..., r

$$\text{DRIF-I}_i(T) = \begin{cases} \text{EST}(T) - \text{EAT}_i & \text{if T needs } R_i, \\ \text{New-EAT}_i(T) - \text{EAT}_i & \text{else.} \end{cases}$$

Recall that EST(T) is the time Task T can begin to use R_i, and that EAT_i tells us when R_i is available. Then the difference between EST(T) and EAT_i is the wasted idle time on R_i if T needs R_i. However, even if T does not need a resource R_i, R_i may be idle if R_i is passive and its EAT_i is less than that of any active resource, such as R_7 in the example of Fig 2. The second case of the formula computes this wasted idle time.

DRIF-N indicates the negative idle times on resources, and is defined as

$$\text{DRIF-N}(T) = \begin{pmatrix} \text{DRIF-N}_1(T) \\ \text{DRIF-N}_2(T) \\ \vdots \\ \vdots \\ \text{DRIF-N}_r(T) \end{pmatrix}$$

where for i = 1, ..., r

$$\text{DRIF-N}_i(T) = \begin{cases} 0 & \text{if T needs } R_i, \\ & \text{or } EAT_i < EST(T), \\ -\text{MIN}(EAT_i - EST(T), C(T)) & \text{else;} \end{cases}$$

The first case indicates that there is no negative idle time for a resource if T needs it, or if it is available before T runs. The second case computes the negative idle times for a resource which is not available at the time T executes. Its negative idle times is defined to be the portion of time, in which the resource is not available while T is running. For example, in Fig 2, R_3 and R_4 have the following negative idle times :

$\text{DRIF-N}_3 = -\text{MIN}(EAT_3 - EST(T), C(T)) = -\text{MIN}(15, 10) = -10$, and

$\text{DRIF-N}_4 = -\text{MIN}(EAT_4 - EST(T), C(T)) = -\text{MIN}(5, 10) = -5$.

DRIF-Q is designed to reflect the "Q" idle times :

$$\text{DRIF-Q}(T) = \begin{Bmatrix} \text{DRIF-Q}_1(T) \\ \text{DRIF-Q}_2(T) \\ \vdots \\ \vdots \\ \text{DRIF-Q}_r(T) \end{Bmatrix}$$

where for i = 1, ..., r

$$\text{DRIF-Q}_i(T) = \begin{cases} 0 & \text{if T needs } R_i, \text{ or } EAT_i > EST(T) + C(T) \\ W_0(EST(T) + C(T) - \text{New-EAT}_i(T)) & \text{else.} \end{cases}$$

There is no "Q" idle time for a resource if it is needed by T, or if it is unavailable during the execution of T. This is the first case in the above formula. In the example of Fig 2, R_1, R_2, R_3 and R_5 all have "Q" idle times equal to zero. The second case deals with resources that are available during the execution of task T. The time duration between the time the resource is available, New-EAT_i, and the time T completes, (EST(T) + C(T)), is the "Q" idle time. Because it is not certain whether in this duration the resource will be used or wasted, we multiply it by a factor W_0 which we choose between 0 and 1. For all the tasks presented here $W_0 = 0.5$.

Then DRIF for task T in Fig 2 is given by

$$\text{DRIF}(T) = \text{DRIF-I}(T) + \text{DRIF-N}(T) + \text{DRIF-Q}(T)$$

$$= \begin{Bmatrix} 5 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 10 \end{Bmatrix} + \begin{Bmatrix} 0 \\ 0 \\ -10 \\ -5 \\ 0 \\ -5 \\ 0 \end{Bmatrix} + \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 2.5 \\ 0 \\ 2.5 \\ 2.5 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 0 \\ -10 \\ -2.5 \\ 0 \\ -2.5 \\ 12.5 \end{Bmatrix}$$

Recall that X_1 = DRUR • DRIF, and DRUR reflects the degree of resource utilization by the tasks remaining to be scheduled. The dot product means that we weight each $DRIF_i$ by $DRUR_i$, and sum them. Therefore, X_1 actually is a sum of *weighted resource idle times*.

To summarize, the heuristic function has been designed to incorporate not only the necessary timing considerations but also resource utilizations. The simulation results shown in the next section indicate that it does work quite well if the appropriate weights W_1, W_2 and W_3 are used.

4. Simulation Results

The purpose of the simulation is to test the performance of the algorithm. Recall that the basic function of the algorithm, as described in Section 2, is to determine a full feasible schedule for a given set of tasks. In each simulation, we input a number of sets of tasks to the local scheduler program. For each of the input task sets it is known exactly how many feasible schedules exist and that each set has at least one feasible schedule because we perform a separate exhaustive search procedure on each set of tasks. We then observe the percentage of task sets guaranteed by our heuristic scheduling algorithm. This percentage is called the *success ratio* of the algorithm.

4.1 Task Generator

The Task Generator is a program that generates tasks for our simulation study. It uses three generating parameters:

1. P1: minimum computation time,
2. P2: maximum computation time, and
3. P3: maximum laxity.

The computation time of a task is randomly chosen between P1 and P2. The deadline of a task is randomly chosen between its computation time and the computation time plus P3. It is clear that different application domains will require different generating parameters.

We assume there are 5 resources, two active resources, and three passive resources. The resource requirements of a task are also chosen randomly with the condition that a task uses at least one active resource.

In the current simulation, we collect 6 independent tasks to form a set. An exhaustive search is made for each set to see how many full feasible schedules exist. For a set of 6 tasks, there are 720 permutations, each of which may or may not represent a full feasible schedule for that task set. We purposely set up the generating parameters used by the Task Generator so that the number of feasible schedules among 720 permutations usually is very small, making it fairly difficult to find a feasible schedule. Table 0 gives the distribution of number of feasible schedules for a typical group of 200 sets of tasks. It shows that among 200 task sets, input to the scheduler simulation program, more than 55% of the task sets (14 + 103 = 117 out of 200) have at most 25 full feasible schedules. And less than 5% of the task sets (6 + 3 = 9 out of 200), have more than 100 full feasible schedules.

Number of Feasible Schedules	1 - 10	11 - 25	26 - 50	51 - 100	101 - 200	201 - 720
Number of Task Sets	14	103	56	18	6	3
Percentage of Sets %	7.0	51.5	28.0	9.0	3.0	1.5

P1 = 30, P2 = 50, P3 = 100

Table 0 : The Distribution of Number of Feasible Schedules among a Group of 200 Sets

4.2 Simulation Results and Observations

The performance of the heuristic algorithm is studied with the sets of tasks for which the task generator has found at least one full feasible schedule. Each simulation is done with a *group* of 200 task sets. In the simulations, we study how the *success ratio*, i.e., the percentage of guaranteed sets, changes with the weights of the heuristic function H, as well as groups. We begin by studying the performance if only one of the three factors in the heuristic function is used in scheduling.

Group Number	1	(no change)	
Weights of function H			
W_1	0	0	1
W_2	0	1	0
W_3	1	0	0
Guaranteed Sets	24	29	25
Success Ratio %	12.0	14.5	12.5

P1 = 20, P2 = 40, P3 = 80

Table 1 : Simulation of an H Function with one Factor

Table 1 shows that, for a given group (group number 1), the scheduling polices corresponding to the "minimum computation time first", "least laxity first" and "least weighted resource idle time first" are not effective. That is, when (W_1, W_2, W_3) = (0 0 1), (0, 1, 0), or (1, 0, 0), the success ratio is lower than 20%. However, for the same group as in Table 1, if we use all three weights and properly tune them, the success ratios can be as high as 80%. This result is shown in Table 2.

Group Number	1	(no change)	
Weights of function H			
W_1	0.20	0.20	0.32
W_2	0.26	0.20	0.21
W_3	0.20	0.20	0.16
Guaranteed Sets	148	152	162
Success Ratio %	74.0	76.0	81.0

P1 = 20, P2 = 40, P3 = 80

Table 2 : Simulation of Full H Function

Group Number	2	3	4	5	6	7	8	9
Guaranteed Sets	170	164	157	174	167	156	176	165
Success Ratio	85.0%	82.0%	78.5%	87.0%	83.5%	78.0%	88.0%	82.5%

P1 = 10 P2 = 30 P3 = 70
W_1 = 0.50 W_2 = 0.36 W_3 = 0.25

Table 3 : 8 groups with fixed generating parameters and fixed weights

Table 3 indicates that with the fixed generating parameters, (P1, P2, P3) = (10, 30 70), and the fixed weights of function H, (W_1, W_2, W_3) = (0.50, 0.36, 0.25), the success ratio varies among 8 specific groups tested from a low of 78% to a high 88%. The range of percentages is due to the randomness of the task generation, but the small standard deviation of about 3%, indicates the stability of the scheduling algorithm. We believe that 78% - 88% is quite a good success ratio. As we will see in Section 5, with some

improvements, the success ratio can be increased.

Group Number	10	11	12	13	14	15
Generating Parameters						
P1	30	20	10	20	20	10
P2	50	60	160	40	100	30
P3	100	120	300	70	100	50
Weights of function H						
W_1	0.35	0.28	0.30	0.32	0.24	0.26
W_2	0.25	0.18	0.19	0.25	0.18	0.20
W_3	0.10	0.13	0.10	0.16	0.14	0.14
Guaranteed Sets	172	160	176	162	164	162
Success Ratio %	86.0	80.0	88.0	81.0	82.0	81.0

Table 4 : Good Weights for 6 Groups

Table 4 lists the best success ratios for an additional 6 groups, each of which is generated by different parameters. The best success ratio for each group is achieved by tuning the weights of function H. We see that in order to achieve the best success ratio, the weights vary with groups, i.e., W_1 varies between 0.24 and 0.35, W_2 varies between 0.18 and 0.25, and W_3 varies between 0.10 and 0.16. It is clear that a set of *universally best* weights do not exist. However, we see that the range of the weights is not very large, suggesting that there may be a set of feasible weights which can cover adequately many groups with different generating parameters. To substantiate this point, we average the weights W_1, W_2, and W_3 in Table 4, and re-use them for the same 6 groups shown in Table 4. Table 5 presents the results:

Group Number	10	11	12	13	14	15
Generating Parameters						
P1	30	20	10	20	20	10
P2	50	60	160	40	100	30
P3	100	120	300	70	100	50
Weights of function H						
W_1	0.29					
W_2	0.21	(no change)				
W_3	0.12					
Guaranteed Sets	170	160	170	159	160	161
Success Ratio %	85.0	80.0	85.0	79.5	80.0	80.5

Table 5 : Using the averaged weights

With a fixed set of average weights, we obtain an average success ratio of more than 80% with a standard deviation of only 2.5%. The results imply that though a set of the universally best weights do not exist, one set of feasible weights may cover many different groups with different generating parameters. Therefore it seems that the weights are not too sensitive to the generating parameters.

Although the simulation results presented in this section are only for a limited number of groups with a limited number of different generating parameters, we have tested many other groups with many different generating parameters. In all cases, the results are quite similar. Due to the lack of space, we are unable to present these additional simulation results here. We would, however, like to point out that these simulation results were in consonance with the observations made above.

5. Extension on the Basic Algorithm

In this section we describe an extension to the basic heuristic scheduling algorithm discussed and evaluated in the previous sections.

5.1 Extended Algorithm

The assumptions underlying the use of the heuristic function in the basic algorithm are

1. At each level of the search, there is a certain *order* among the tasks to be selected;
2. The order can be *identified* by a linear function such as function H used in our primary algorithm.

Though the first assumption is definitely true, the second may not always hold, so our original algorithm cannot always guarantee a set of tasks for which there is at least one full feasible schedule. To improve the success ratio, the following means were considered:

1. add some non-linear components to function H;
2. change the weights of function H dynamically;
3. whenever a partial non-strongly feasible schedule is met while scheduling, try to backtrack.

Though the first two alternatives are more mathematical than the third, we did not adopt them because the first increases the computation cost on every computation of function H,

and the second could make the algorithm too complex. The third method was adopted because we made an interesting observation from the log files of the primary algorithm runs: If, at the point where a non-strongly feasible schedule is met, the task with the second minimum value of the function H was selected instead of the minimum one, more than half of the task sets that were not scheduled by the primary algorithm become schedulable. Moreover, a majority of the remaining task sets failing to be scheduled become schedulable if we backtrack to an ancestor and select the task with the second minimum value of function H at that level, instead of the minimum one. That is, function H not only tells us which is the best task to be selected, but also gives information about which one is the next best if the best does not work.

With this observation, the primary algorithm, given in Fig 1, is extended in the following way: Each scheduled task has a pointer to the task with the second minimum value of function H, and it also records the old EAT values before it is scheduled. This is done so that the "second pointer" and old EAT values may be used by possible future backtracks without re-calculations at that level. Each time a non-strongly feasible schedule is found, a subroutine, called the Limited-Backtracker, is invoked to

1. withdraw the task just selected and added in the schedule,[1] and instead attempt to schedule the task with the second minimum value of function H;

2. if the first step does not succeed, that is, the schedule is still non-strongly feasible, recursively backtrack to the immediate ancestor and attempt to schedule the task with the second value of function H at the ancestor level. Whenever a strongly

1. Note that the schedule is feasible, but not strongly feasible, therefore the task withdrawn can be guaranteed, but some others will not.

feasible schedule is found, the Limited-Backtracker returns "success" to the caller, the scheduler program. Otherwise, it continues the recursive backtrack until either it has backtracked to the root of the search tree — the empty schedule, indicating that all the ancestors have been tried; or until a counter, which counts the number of backtracks in scheduling this task set, reaches a pre-set upper bound. In these cases, the Limited-Backtracker returns "failed".

The pseudo code of the algorithm for the Limited-Backtracker is shown in Fig 3. We call the first step in the Limited-Backtracker a *pseudo backtrack* because it happens at the current search level and function H is not recalculated. The second step is called *real backtrack*. Real backtracks do increase the computation cost because they requires the re-calculations of function H at (all) the level(s) (in search three) immediately below the vertex in which the real backtrack succeeds.

Note that, if in the Limited-Backtracker we did not limit the number of real backtracks, then in the worst case, the search process might eventually expand two vertices from each ancestor, resulting in a computation time proportional to 2^n, where n is the number of tasks.

```
Procedure Limited-Backtracker(VAR task-set : task-set-type; VAR schedule : schedule-type);

(* Limited-Backtracker is called by the scheduler when the (partial) schedule is found
   non-strongly feasible. Task-set contains the tasks remaining to be scheduled;
   schedule is the non-strongly feasible schedule.                    *)

BEGIN

  ("first, we do the pseudo backtrack")
  let T1 be the last task in the schedule;
  remove T1 from the schedule;
  let T2 be the task with the second H value pointed to by the "second pointer" of T1;
  task-set := task-set + {T1} - {T2};
  put T2 at the end of the schedule;
  IF strongly-feasible(schedule) THEN
     return(success)
     ELSE

     ("the real backtrack starts")
     WHILE NOT empty(schedule) and counter < max-counter DO
       BEGIN
         ("withdraw from the end of the schedule all the tasks
           with a second minimum value of function H.
           This kind of tasks have their "second pointer" nil.")
         REPEAT
           let T1 be the last task in the schedule;
           remove T1 from the schedule;
           task-set := task-set + { T1 };
         UNTIL T1's "second pointer" <> nil;

         EAT := old-EAT stored with T1;

         let T2 be the task pointed by T1's "second pointer";
         task-set := task-set - {T2};
         put T2 at the end of the schedule;

        if strongly-feasible(schedule) then
           return(success);

        counter := counter + 1;
     END("WHILE");

  return(failed);
END;
```

Fig 3 : The Algorithm of the Limited-Backtracker

5.2 Simulation Studies on the Extended Algorithm

The results of simulations for the extended scheduling algorithm are shown in Table 6 and 7.

Group Number	20	(no change)	
Generating Parameters			
P1	30	(no change)	
P2	60		
P3	100		
Weights of function H			
W_1	0.20	0.26	0.24
W_2	0.28	0.20	0.20
W_3	0.20	0.16	0.14
SR_0 %	72.5	77.0	81.0
SR_1 %	90.0	94.5	95.5
SR_2 %	95.0	98.0	99.5
Max Real Backtracks	6	7	6

SR_0 : the basic success ratio without any backtrack
SR_1 : the success ratio without any real backtrack
SR_2 : the success ratio with pseudo and real backtracks

Table 6 : Simulation Results of the Scheduling Algorithm with Limited Backtracking

From Table 6, first, we can see that though the success ratios without backtracks (SR_0) varies with different sets of weights from 72.5% to 81%, using pseudo backtracks a success ratio (SR_1) of between 90% and 95% results. The final success ratio with pseudo and real backtracks (SR_2) is no less than 95% and can be as high as 99.5%. Thus, though the scheduling costs increase with the use of backtracking, the success ratio increases substantially.

Secondly, we may notice that this simulation is one group with three fairly different sets of weights. In all three cases, the final success ratios are surprisingly high. This indicates that with backtracking, tuning of weights is less important than in the primary algorithm. This is because backtracking is more resilient in the sense of allowing wrong choices to be rectified.

Table 7 lists the results of simulation with a set of fixed weights for 5 different groups over a range of generating parameters. We observe that with limited backtracking, a set of common weights is not only feasible, but is also sufficient. The success ratio is higher than 96% if limited backtracks are allowed.

Group Number	21	22	23	24	25
Generating Parameters					
P1	20	30	20	20	50
P2	50	50	60	120	200
P3	70	100	80	150	300
SR_0 %	81.5	75.0	82.5	77.5	72.0
SR_1 %	96.5	90.0	91.5	94.0	92.5
SR_2 %	99.5	97.5	96.0	99.5	98.0
Max Real Backtrack	2	7	11	5	5

SR_0, SR_1, and SR_2 have the same meanings as in Table 6
$W_1 = 0.26$, $W_2 = 0.20$, $W_3 = 0.14$

Table 7 : One Set of Weights for Different Groups

Tables 6 and 7 also list the maximum numbers of real backtracks used for each group of 200 task sets. Note that usually no more than 10% of task sets, (SR_2 - SR_0), use real backtracks. The numbers of real backtracks used by those task sets often are very small. Tables 6 and 7 indicate that even the maximum number of real backtracks for a

group usually is less than 10. Thus, the use of the extended algorithm with limited backtracking does not increase the scheduling cost appreciably.

6. Complexity and Optimality

In this section we discuss the time complexity of the algorithm, propose several metrics of optimality, and illustrate how well our algorithm performs in terms of these optimality measurements.

6.1 Time Complexity

The major computation happens at the calculations of function H. Each calculation of function H takes time proportional to r. Function H is invoked (Σ i (i=1, ..., n)) times, resulting in the total time complexity of rn^2. Pseudo backtracks do not increase the computational complexity. However, the computation cost increased by real backtracks cannot effect the total complexity, so long as the upper bound of real backtracks is pre-set to less than n^2. Our simulation results in Section 5 showed that the number of real backtracks was usually quite low. The time complexity rn^2 of our heuristic scheduling algorithm is fairly low compared to that of an exhaustive search algorithm which takes time proportional to rn!.

6.2 Optimality

Before discussing optimality, we need to chose some metrics suited for hard real-time environments. For a full feasible schedule we propose the following metrics :

1. *Task Set Completion Time* : This is the time when, according to the full feasible schedule, all tasks can be completed. Thus it reflects the system response time.

2. *Average utilization ratio of resources* : This is the average of resource utilization ratios of the tasks in the full feasible schedule. This is one of the traditional metrics of optimality for scheduling algorithms.

3. *Average laxity* : This is defined as the average of the differences between deadlines and scheduled completion times of all tasks in the full feasible schedule. This is a metric indicating the average laxity of completed tasks.

4. *Minimum laxity* : The minimum of differences between deadlines and scheduled completion times of all tasks in the full feasible schedule. We can postpone executing tasks by this amount so that a newly arriving task may be executed before them.

For each set of tasks input to the scheduler simulation program, using exhaustive search, we calculate the following:

1. the values of the four metrics for each permutation that represents a feasible schedule; and

2. the optimal, average, and worst values of each metric among all permutations representing feasible schedules.

With respect to each metric, the full feasible schedule determined by our extended heuristic algorithm is ranked as follows:

1. *optimal*, if its performance under that metric is equal to the optimal one found above;

2. *good*, if its performance is worse than the optimal but better than or equal to the average value;

3. *poor*, if its performance is worse than the average, but better than the worst;

4. *worst*, it its performance is equal to the worst.

In some cases, the optimal and worst values of a metric for a set of tasks are equal. Then, if the scheduler guarantees the set of tasks, the full feasible schedule will have the same value of the metric as the optimal as well as the worst one. Such a full feasible schedule is termed *optimal'*.

Simulation results for one group of 200 sets of tasks are shown in Table 8. In that simulation, the final success ratio is 98.5%, therefore total 197 sets of tasks are guaranteed and their full feasible schedules are ranked according to the optimality metrics above.

measurements	optimal'	optimal	good	poor	worst
Set Completion Time	137	25	6	7	22
Average Utilization	97	15	80	4	1
Average Laxity	42	37	45	45	28
Minimum Laxity	94	85	3	6	9

P1 = 10	W_1 = 0.36	Input sets of tasks	= 200
P2 = 30	W_2 = 0.25	Success Ratio %	= 98.5
P3 = 70	W_3 = 0.11	Compared sets of tasks	= 197

Table 8 : Comparisons of Optimalities

From Table 8, we see that the full feasible schedules found by the scheduler are optimal or close to optimal most of the time. For example, using average resource utilization as a metric, we found that in 97 + 15 = 112 cases out of 197, our algorithm found the optimal solutions, and another 80 out of 197 are close to optimal (that is, good). Simulations and comparisons made for many other groups had similar results. We believe that these results indicate that by developing a good heuristic function and tuning the weights properly one can expect extremely good results from this approach.

7. Conclusions

The problem of determining an optimal schedule is known to be NP-hard [GRAH79] and is hence impractical for real-time task scheduling. The problem is further complicated when, in addition to computation times and deadlines of tasks, their resource requirements are also accounted for.

In this paper, we described a heuristic approach to solving this problem. The crux of our approach lies in the heuristic function used to select the task to be scheduled next. It uses information about tasks' real-time constraints and resource utilizations. Simulation studies showed that the weights for the various factors that affect the scheduling decisions have to be fine-tuned in order to obtain a degree of success comparable to that obtained via exhaustive search. However, use of limited backtracking, whereby in case of a failure, previous scheduling decisions are rescinded and new decisions are made, was observed to improve the degree of success substantially. This improvement was observed even when the initial set of weights were not tailored for a particular set of tasks. Simulation studies also showed that in most cases the full feasible schedule determined by the heuristic algorithm was optimal or close to optimal.

Given that the heuristic approach to task scheduling with resource constraints has been highly successful, we plan to incorporate it in our distributed scheduling algorithm. This will involve extensions to the basic bidding technique [RAMA84]. In particular, modifications have to be made regarding the information that is sent on bids so that a requesting node can evaluate bids and determine the best bidder.

8. References

[DERT74] Dertouzos, M., "Control Robotics: The Procedural Control of Physical Process", *Proc. of the IFIP Congress*, 1974.

[EFE82] Efe, Kemal, "Heuristic Models of Task Assignment Scheduling in Distributed Systems", *IEEE Computer*, June 1982.

[GARE75] Garey, M.R. and Johnson, D.S., "Complexity results for Multiprocessor Scheduling under Resource Constraints", *SIAM Journal of Computing*, 4, 1975.

[GRAH79] Graham, R.L. et al, "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey", *Annals of Discrete Mathematics*, 5, 1979.

[JOHN74] Johnson, H. H. and Madison, M.S., "Deadline Scheduling for a Real-Time Multiprocessor", NTIS (N76-15843), Springfield, VA, May 1974.

[LEIN80] Leinbaugh, D. W., "Guaranteed Response Times in a Hard Real-Time Environment," *IEEE Transactions on Software Engineering*, Vol. SE-6, Jan. 1980.

[LEIN82] Leinbaugh, D.W. and Yamini, M., "Guaranteed Response Times in a Distributed Hard-Real-Time Environment", *Proc. of the Real-Time Systems Symposium*, Dec. 1982.

[LEN82] Lenat, Douglas B., "The Nature of Heuristics", *Artificial Intelligence*, 19 (1982).

[LEN83] Lenat, Douglas B., "Theory Formation by Heuristic Search — The Nature of Heuristics II: Background and Examples", *Artificial Intelligence*, 21 (1983).

[LIU73] Liu, C. L., and Layland, J., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, Jan. 1973.

[MA82] Ma, Richard P., Lee, Edward Y.S., and Tsuchiya, Masahiro, "A Task Allocation Model for Distributed Computing Systems", *IEEE Transactions on Computers*, Vol C-31, No. 1, Jan. 1982.

[MA84] Ma, Richard P., "A Model to Solve Timing-Critical Application Problems in Distributed Computer Systems", *IEEE Computer*, Jan. 1984.

[MOK78] Mok, A.K. and Dertouzos, M.L., "Multiprocessor Scheduling in a Hard Real-Time Environment", *Proc. of the Seventh Texas Conference on Computing Systems*, Nov 1978.

[MUNT70] Muntz, R. R., and Coffman, E. G., "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems," *Journal of the ACM*, Vol. 17, No. 2, April 1970.

[RAMA84] Ramamritham, Krithivasan, and Stankovic, John A., "Dynamic Task Scheduling in Distributed Hard Real-Time Systems", *Proceeding of 4th International Conference on Distributed Computer Systems*, May 1984.

[SCHO84] Schoeffler, J.D., "Distributed Computer Systems for Industrial Process Control", *IEEE Computer*, Vol 17, No. 2, Feb. 1984.

[TEIX78] Teixeira, T., "Static Priority Interrupt Scheduling", *Prof. of the Seventh Texas Conference on Computing Systems*, Nov. 1978.

6.0 Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies
Commander USAERADCOM ATTN: DELSD-L-S Fort Monmouth, NJ 07703	2 copies
Commander USACECOM ATTN: DRSEL-COM-RF (Dr. Klein) Fort Monmouth, NJ 07703	1 copy
Commander USACECOM ATTN: DRSEL-COM-D (E. Famolari) Fort Monmouth, NJ 07703	1 copy
Commander USACECOM ATTN: DRSEL-COM-RF-2 (C. Graff) Fort Monmouth, NJ 07703	10 copies

END

FILMED

12-84

DTIC