THE INTEGRATION OF DEADLINE AND CRITICALNESS IN HARD REAL-TIME SCHEDULING

Sara R. Biyabani, John A. Stankovic and Krithi Ramamritham

COINS Technical Report 88-82

The Integration of Deadline and Criticalness in Hard Real-Time Scheduling[1]

Sara R. Biyabani[2]
Dept. of Electrical and Computer Engineering

John A. Stankovic
Krithi Ramamritham
Dept. of Computer and Information Science

University of Massachusetts

Abstract

Two task scheduling algorithms for distributed hard real-time computer systems are presented. Both scheduling algorithms are based on a heuristic approach since an exact solution is computationally intractable. The algorithms explicitly account for both the deadlines and criticalness of tasks when making scheduling decisions. In analyzing the new algorithms, a performance metric called the **Weighted Guarantee Ratio** is defined. This metric reflects both the percentage of tasks which make their deadlines as well as their relative worth to the system. The performance analysis is done by simulating the behavior of the algorithms as well as that of several other pertinent baseline algorithms under a wide range of system conditions including a non-homogeneous task arrival rate. The results show that the algorithms outperform all the baseline algorithms - except for the ideal but impractical, centralized baseline - under the range of system conditions studied, and that in many cases they perform close to the ideal.

1 Introduction

Hard real-time systems are those systems in which the correctness of the system depends on both the *logical result* of the computation as well as the *time* at which such a result is produced. In many hard real-time systems it is crucial for the tasks in the system to meet their specified *deadlines*, otherwise the tasks are worthless, or worse, cause catastrophic results. This strictness in meeting deadlines makes scheduling an important issue in the correctness and reliability of the system.

In the past, many *static* task scheduling algorithms have been developed for hard real-time systems, where the calculation of schedules has been done off-line using complete knowledge about the tasks' characteristics and arrival times. These algorithms have low run-time costs but they often fail to adequately adapt to changes in the environment (especially overloads), they are expensive to modify as the system evolves, and they seem to be most suitable to relatively small systems. Complex real-time systems require more adaptive solutions based on *dynamic* scheduling algorithms. Such algorithms plan task schedules at run-time and can therefore better respond to changes in the environment and better evolve over time, but with increased run-time costs and with the possibility of missing some deadlines. Since some deadlines might be missed, e.g., due to overloads, unexpected combination of events, or failures, we require algorithms that guarantee that the more critical tasks will make their deadlines. This paper is concerned only with such dynamic hard real-time scheduling algorithms.

A characteristic of most previous real-time scheduling algorithms is the use of priority based scheduling. Here tasks are assigned 'priorities', which are implicit or explicit functions of their deadlines or *criticalness* or both. (The criticalness of a task is an indication of its level of importance.) However, in actuality, these two requirements sometimes conflict with each other. That is, tasks with very short deadlines might not be very critical, and vice versa. This causes a dilemma in choosing the appropriate priority (i.e., in choosing the appropriate single value to represent two separate concerns). We propose and analyze two distributed, dynamic scheduling algorithms, ALG1 and ALG2. They avoid the dilemma of priority scheduling, yet integrate criticalness and deadline such that, not only do the more critical tasks meet their deadlines, but many other less critical tasks also meet their deadlines. Overall, their goal is to maximize the net worth of the executed tasks to the system.

Note that a dynamic scheduling algorithm cannot ensure that all tasks will meet their deadlines. Thus, they may not be applicable to a specific type of task known as a *safety critical* task. If any safety critical task misses its deadline, catastrophic consequences may result. These tasks should be dealt with via schemes such as preallocation of resources. Dynamic scheduling algorithms such as those discussed in this paper, are applicable to *essential* and *non-essential* tasks. Missing deadlines of essential tasks will seriously degrade the functioning of the system, but not cause any catastrophes. Non essential tasks are of less value than essential tasks to the system, yet under most conditions should also execute by their deadline. It is necessary to meet the deadlines of as many essential and non-essential tasks as possible.

Section 2 examines related work. In Section 3, we describe the characteristics of the assumed hard real-time distributed system. The details of the algorithms, ALG1 and ALG2, are outlined in Section 4, followed by an evaluation of the performance of the algorithms in Sections 5 and 6. The performance analysis is done by simulating the behavior of the algorithms as well as that of several other pertinent baseline algorithms under a wide range of system conditions including a non-homogeneous task arrival rate. Section 7 discusses and summarizes the results.

2 Related Work

Many real-time scheduling algorithms have been studied in the past few decades, and most have been shown to be computationally intractable, e.g., [GJ, 75]. An exhaustive cataloging of all real time scheduling algorithms proposed for central and multi-

[1]This work was partly supported by the Office of Naval Research under contract 048-716/3-22-85.

[2]This paper was produced before this author joined Digital Equipment Corp. The views expressed are exclusively those of the author and do not reflect the opinions or future product plans of Digital.

processor systems is beyond the scope of this paper. [Cheng, 87] contains an extensive survey of scheduling algorithms in this domain. We will present only a brief overview of some of the major scheduling algorithms proposed in the past for systems with hard real-time constraints. We note that some of the earlier work done on scheduling in central and multiprocessor systems has greatly influenced scheduling in current distributed systems. Examples include [GJ, 76], [LL, 73], [JLT, 85], [MD, 78], [Mok, 83], [Chu, 83], [Stan, 85], and [SLR, 86]. Some of this referenced work applies to static scheduling and some to dynamic scheduling. As mentioned above, dynamic scheduling schemes tend to have more flexibility and adaptibility than the static ones, since the decisions as to *whether* to execute a task and *when* to execute are made at run time. Studies on dynamic scheduling in distributed systems also involve issues of load balancing. Many algorithms for load balancing in non-hard real time systems have been reported, but they do not explicitly take into account task deadlines. However, the work on dynamic scheduling in distributed hard real-time systems that has been carried out under the Spring Project has considered deadlines ([RS, 84], [ZRS, 87], [ZRS, 87a], and [SRC, 85a]). The algorithms described here are an extension of this work.

The real-time literature shows that most previous scheduling work has been primarily concerned with either strictly deadline driven or priority based schemes. Very little work has been done in the area of scheduling tasks with hard real-time constraints, and different levels of criticalness. One study that has attempted to combine the deadline based and priority based approaches is [CL, 85]. The concepts in this study are good abstractions, but heuristics are needed before they can be applied in practice. Also, the study assumes that tasks may miss their deadlines, but that such tasks continue to execute. This makes the algorithm applicable to *soft* real-time systems.

Other real-time scheduling schemes translate deadlines into 'priorities' and then use the task's priority as the criterion to determine schedulability. This assumes that the higher the criticalness of a task, the earlier its deadline. Further, the only way to give one process a higher priority than another is to give it a shorter deadline. However, in reality, some low criticalness tasks may have earlier deadlines and some high criticalness tasks, larger deadlines. The problem is to formulate a real-time task model which allows a distinction between these two attributes. This is what our algorithms do.

3 The System Model

In this section, we briefly discuss the various parameters of the distributed system and the tasks that are to be scheduled. Section 3.1 concentrates on the characteristics of the application tasks in the system. Section 3.2 discusses the semantics of the guarantee. Section 3.3 discusses the system architecture.

3.1 The Characteristics of An Application Task

Our assumptions regarding the application tasks are as follows. All application tasks are known. Their invocation order is not known. That is, tasks arrive dynamically and independently. At run-time there is no a priori knowledge of which tasks will arrive and when they will arrive. There are no precedence constraints on the tasks; they can be run in any order relative to each other as long as their deadlines are met. All tasks are aperiodic and there are no stipulations about their earliest start time; a task is ready to execute as soon as it arrives in the system.

Each task has the following characteristics :

- an arrival time: the time at which the task is invoked,
- a worst-case computation time: the maximum time needed for it to complete execution,
- a criticalness: one of the n possible levels of importance of the task (in this study, $n = 10$), and
- a deadline: the time by which the task has to complete execution.

These are the static attributes of a task and are known at the time of arrival of the task. It should be noted that in the evaluation reported here, the characteristics of the task are time invariant; for example, if a task has criticalness 9 when it arrives in the system at time, I_0, then it's criticalness is not dynamically changed. However, such a change can easily be handled by our approach.

3.2 Semantics of the Guarantee

When all tasks have equal criticalness values, a task is said to be guaranteed if the scheduling algorithm can certify that the task will complete execution before its deadline in spite of future arrivals. This implies that newly invoked tasks can be guaranteed provided previously guaranteed tasks are not jeopardized.

The semantics of guarantee poses a problem when tasks with differing criticalness values are present. Suppose a task has been guaranteed and a task with a higher criticalness arrives. Also suppose that the new task can be guaranteed only if the lower criticalness task is removed from the guaranteed list, i.e., the guarantee is withdrawn. In this case the initial guarantee is not absolute but conditional upon the non arrival of higher criticalness tasks which conflict with it. This is the semantics associated with the term guarantee used in this paper. In most applications it is important to meet the deadlines of higher criticalness tasks even if that implies the withdrawal of guarantees to other (lower criticalness) tasks.

Note that the characteristics assumed for tasks in the system are important in performing the dynamic guarantee. For example, with the assumptions concerning tasks given in Section 3.1, a relatively simple guarantee algorithm can be used, as is found in [RS, 84], [SRC, 85a]. We have also developed more sophisticated guarantee algorithms that integrate cpu scheduling and resource allocation [ZRS, 87], [ZRS, 87a]. That is, tasks can be preemptive or non-preemptive and they may require additional resources besides the cpu. The algorithms described in this paper could also be applied to these other more sophisticated guarantee algorithms. However, in order to concentrate on the criticalness issues, in this paper we study two algorithms that assume the simpler task characteristics.

3.3 The System Architecture

In this work, we study two task scheduling algorithms for a distributed system consisting of N nodes. Each node contains m processors divided into two types; *system processors* are dedicated to executing system tasks and *application processors* execute only application tasks[3]. The connection medium for the nodes is assumed to be a shared (broadcast) bus. Hence, the

[3] All the performance data presented in this paper assumes a single system processor and a single application processor.

distributed system under consideration is a collection of multiprocessors connected together in a loosely-coupled network.

The main system tasks of interest to the discussion in this paper are the local scheduler and the global scheduler. The local scheduler at each node maintains a data structure called the *System Task Table*, STT. This table contains a list of application tasks that have been dynamically guaranteed to make their deadline at this local node. Entries in the STT are arranged in the order of execution and tasks are dispatched for execution from this table. Each STT entry, corresponding to a guaranteed task, has five attributes: the arrival time, the latest start time (computed when the guarantee is issued), the criticalness, the deadline, and the computation time. Except for the latest start time, all the attributes are inputs. The latest start time is the latest time in which the task can start executing and still meet its deadline.

The Local Scheduler, which can re-order, insert or remove any entries in the STT, is activated upon the arrival of a new task at the node, or in response to the bidding which is initiated by the global scheduler. The Local Scheduler, working on a copy of the STT, determines if a new task can be inserted into the current STT such that all previous tasks in the STT as well as the new task meet their deadlines. If so, we say that the task is guaranteed. The latest start time of a task is determined when it is guaranteed. The details of the local guarantee have appeared elsewhere [SRC, 85a], and due to space limitations are not repeated here. If the new task cannot be guaranteed locally, or can only be accomodated at the expense of some previously guaranteed task/s, then the rejected task/s are handed over to the Global Scheduler.

The Global Scheduler then takes the necessary actions to transfer the task/s to any alternative nodes that may have the resources to accept those tasks. The Global Scheduler uses bidding. Request-for-bids (RFB) are broadcast to the other nodes when a local task has to be reallocated. When an 'adequate' number of remote nodes respond with bids reflecting their surplus, the Global Scheduler evaluates those bids and transfers the task to the node with the best bid. Also, there is a window of time, *the time-out period*, during which the bids are accepted for a given task. The Global Scheduler also makes bids in response to RFB's from other nodes. The bidding algorithm was proposed by [RS, 84] and has been used by [ZRS, 87] and [Cheng, 87]. That same bidding algorithm is also used here. Note that the overall cost of bidding and moving a task can be reduced by having copies of the code for tasks at selected nodes, and in some cases just signalling an activation of the remote copy, or in other cases transferring only state information for the task rather than the task code itself.

4 The Algorithms

We determine the schedulability of an incoming task as quickly as possible after its arrival[4]. Both of the new algorithms first attempt to guarantee an incoming task according to its deadline, ignoring its criticalness. If the task is guaranteed then the scheduling is successful.

However, if this first attempt at scheduling fails, then there is an attempt to guarantee the new task at the expense of previously guaranteed, but less critical tasks. If enough less critical tasks can be found then the new task is guaranteed at this site and the removed tasks are transferred to alternative sites. If there are not enough less critical tasks, or the deadline of the new task is such that the removal of any such tasks does not allow the new task to meet its deadline, then the *new task* is transferred to an alternative site. The process is repeated at the next node until the task either meets its deadline or its deadline expires.

The two algorithms differ only in how they remove low criticalness tasks from the STT. In algorithm ALG1, lower criticalness tasks are removed one at a time and in strict order from low to high criticalness. Algorithm ALG2 also only removes tasks of lower criticalness, but does not follow the strict order found in the first algorithm. Rather, it removes any task with lower criticalness, starting from tasks with the largest deadline. Since the algorithms are so similar we present the two algorithms, noting the differences.

4.1 The Algorithms

In this subsection, we describe the two algorithms as viewed from the perspective of a new task, T_{new}.

Step 1 Determine the schedulability of a task, T_{new}, based on *its deadline and the deadlines of tasks* in the STT. Guarantee the task if it can be inserted into the STT such that T_{new} and all previously guaranteed tasks can all meet their deadlines. Again, refer to [SRC, 85a] for details on the guarantee algorithm.

Step 2 If Step 1 fails, attempt to guarantee the new task by removing a sufficient number of previously guaranteed tasks in this manner:

- Define the *Window* of T_{new} to be that portion of the STT from the deadline of T_{new} forward to time zero. That is, all tasks in the STT scheduled after the deadline of the new task are excluded from consideration because their removal cannot help in scheduling the new task.

ALG1 Temporarily remove the task with the lowest criticalness level, i, *within the Window of* T_{new}, and if there is more than one task with this criticalness, the task with the largest deadline.

ALG2 Temporarily remove the first task within the Window of T_{new} such that its criticalness is less than T_{new} and its deadline is the largest among all the tasks in the Window.

- If any task is removed then try to guarantee T_{new} again, based on deadlines as outlined in Step 1. Repeat the removal process until either T_{new} is guaranteed or no lower criticalness tasks remain in the window.
- If T_{new} is not guaranteed reallocate it to another node. Note, in this case, any task that might have been temporarily removed is restored. If T_{new} is guaranteed, reallocate the removed task(s) to another node.

In the above algorithms, re-guaranteeing on each task removal may be expensive. Various heuristics are possible here. For example, for the results presented in this paper, we use a very simple heuristic as follows: If the sum of the computation time(s) of the task(s) removed so far is equal to or greater than the computation time of T_{new}, then at that time T_{new} is re-guaranteed. This

[4] This approach has many advantages as detailed in [RS, 84] and [Stan,87].

scheme is inexpensive and works because of the task assumptions made in this paper. On the other hand, it is pessimistic because it may preempt more tasks than is necessary by not accounting for already available free time. Since we are testing overloads (and nodes have very high utilizations) there is little free time so this approximation works well. In general, when tasks require more resources than the cpu and have future arrival times (e.g., periodic tasks), one would have to use a full re-guarantee procedure after each task removal.

In summary, both tasks try to guarantee a new task without removing previously guaranteed tasks. If this is not possible, ALG1 removes the least critical tasks whereas ALG2 removes any *lower* criticalness tasks with long deadlines. This difference in the algorithms results in the deadlines of tasks reallocated by ALG2, on the average being longer than the tasks reallocated by ALG1. Hence, it can be said that, on an average, ALG1 attempts to reallocate more low criticalness tasks with tighter deadlines than ALG2. This could then cause the reallocated tasks to have a lower probability of being guaranteed under ALG1 than under ALG2. But, ALG2, on the average, attempts to reallocate higher criticalness tasks. So if it is not successful, then there is more impact on the value to the system.

5 Evaluation of the Algorithms

We evaluate our algorithms via simulation. The system modeled consists of 5 nodes attached to a common bus. In this paper, we report on how the overall system load, the laxity, and the distribution of the system load among the nodes, affect the performance of the algorithms. The *laxity* of a task is defined as (Deadline - Computation time). Due to space considerations, only representative performance results are shown for each of the above areas. The effects of variations in the task-dependent parameters, of variations in weighting schemes, and of varying arrival distributions of tasks at different *criticalness levels* were also studied and are reported in [Biya, 88].

5.1 Performance Metric

One metric used in real-time scheduling is to determine the percentage of tasks which make their deadline. Call this the guarantee ratio, GR. We do state the GR metric in a few places in the paper. However, a more relevant metric in a system that has tasks of unequal criticalness is to guage the percentage of tasks of each criticalness level that are guaranteed. This value is dependent on the weights attached to the tasks at each level of importance. The determination of which tasks should be attached what specific weights is highly dependent on the particular application environment. We did test a few weighting schemes, such as linearly or exponentially increasing weights.

Define: Weighted Guarantee Ratio, $WGR = \sum_i WGR_i$

$$WGR = 100 * \frac{\sum_i (c_i * T_G)}{\sum_i (c_i * T_C)}$$

where

i = criticalness level.

c_i is the weight of the criticalness level, the default is

$$c_i = e^{i-1}.$$

T_G is the total number of tasks guaranteed at that criticalness level.

T_C is the total number of tasks created (generated) at that criticalness level.

5.2 Baselines

In order to understand the performance of our new algorithms, we need to compare and contrast them with algorithms that are often used or, are upper and lower bounds. We are especially interested in comparing our algorithms with those using only deadlines or criticalness. Comparison with these should demonstrate advantages, if any, of combining deadline and criticalness in determining the schedulability of hard real-time tasks.

We compare the performance of our 2 new algorithms with each other and with that of four other baseline algorithms under identical conditions. Two of these baselines schedule based on deadlines, one schedules based on criticalness, and yet another schedules based on a combined (deadline and criticalness) criterion that assumes perfect state knowledge. The baselines are:

- **DDLN**, tasks are guaranteed based on deadline only, and if not guaranteed locally a distributed algorithm is run (the same one as for ALG1 and ALG2). This algorithm is exactly like ALG1 and ALG2 except that a task that has been inserted into the STT is not removed - irrespective of the criticalness of the task.
- **NC_CR**, tasks are scheduled "highest criticalness first" but without any node cooperation. There is no notion of a guarantee in this algorithm.
- **NC_DD**, tasks are scheduled "earliest deadline first," but without any node cooperation. There is no notion of a guarantee in this algorithm.
- **C_CRDD**, this is a centralized non-preemptive resume algorithm. It assumes that all the tasks in the system arrive at a single queue ordered according to criticalness. Tasks with the same criticalness are ordered based on their deadlines. Tasks are assigned to processors on a FCFS basis. There is no guarantee and this algorithm simply serves as an upper bound on weighted success ratio.

Except for the case of DDLN, we assume that the time required to schedule tasks in all the baseline algorithms is zero, i.e., the overheads due to scheduling time are nil. Hence, the results obtained for the last three baseline algorithms represent their respective ideal cases. It should be noted that we do simulate the scheduling (and remote node re-scheduling) time overheads for the algorithms, ALG1 and ALG2, as a function of the number of tasks at a node when the algorithms are invoked. Specifically, we account for the time taken to manipulate the various linked lists, including the STT, maintained by the scheduler. The assumed overheads are: 5 microsecs for traversing a link, and 10 microsecs for deleting or inserting an element. Hence, the results obtained for these proposed algorithms are more realistic under the given system conditions than are the ideal ones for the last three baselines.

6 Experimental Parameters

We now discuss the results. Since there are five nodes in the

system, the total system load is the sum of the load on each of the five nodes. The simulation parameters, R and r_n, denote the total system load and the load on node n, respectively, i.e.,

$$R = \sum_{n=1}^{5} r_n.$$

For a given system load, the external arrival load on *each of the nodes* could be distributed in quite different ways. All the nodes could have exactly the same load, or totally uncorrelated loads, or a linearly, or even exponentially, increasing load relative to each other, but always summing to the same total system load. The effects of varying the network load distribution indicate the adaptiveness of the global component of our algorithms to different network load patterns and hence, are important to study in evaluating their performance.

We define a simulation parameter, b, the balance factor [ZRS, 87] to denote the balance of load - with respect to arrival of tasks - across the nodes. A high value of b implies a more balanced system, whereas a low value of b implies a less balanced system. The value, 1.0, signifies a completely balanced system; the arrival rate is the same at each of the five nodes. On the other hand, a value of $b = 0.3$ signifies an unbalanced system where the busiest node has a load of r_B and the next most busy has a load of $(0.3 * r_B)$ and the next most busy, $(0.3 * (0.3 * r_B))$ and so on.

Therefore, for a given load, R and a given value of balance factor, b, there is unique load pattern for the nodes in the system. The following gives the relationship between system load, R, the balance factor, b, total number of nodes in the system, N, and the load on the busiest node, r_B :

$$R = \sum_{n=1}^{N} (b^{n-1} * r_B)$$

Throughout all studies, the task computation time was sampled from an exponential distribution with a mean of 100ms. The laxity was also sampled from an exponential distribution, but with means that were multiples of the mean of the computation time. Thus, for average laxities of 3 and 6 the laxity was derived from exponential distributions with means of 300ms and 600ms, respectively.

As for the relative number of tasks in each of the 10 criticalness classes allowed, the distribution was uniform. That is, at each node there were equal percentage (roughly 10%) of tasks generated in each of the 10 criticalness classes. This uniform distribution of tasks in different criticalness classes is to be distinguished from the distribution of load across the five nodes in the system.

6.1 System Load and Laxity

In this section we present the results of varying the system load and the laxity. The other system parameters are held constant. The performance of our algorithms and that of the baselines is compared for loads R = 2.5, 5, 10 and 15, when the mean laxity of all tasks in the system is 3 times the mean execution time of the tasks. This range of values was chosen to cover conditions from light to heavy loads. To test the load balancing aspects of our algorithms and the baseline algorithms, we set b = 0.5. In other words the system load is not balanced: each node has half the load of the previous node.

Observations and Discussion:

In Figure 1 the WGR's for our two algorithms as well as for those of the baseline algorithms, DDLN, NC_CR, NC_DD and C_CRDD, are plotted against the system load for b = 0.5 and average laxity 3.

From Figure 1 we see that all algorithms deteriorate in performance with respect to WGR with an increase in load. The highest values for WGR are given by the centralized, C_CRDD algorithm. This is expected since C_CRDD orders all tasks in the system in a single queue and gives preference to tasks of high criticalness - all at zero cost.

Except for the case of the highest load, $R = 15$, the worst performance is given by the non-cooperative deadline based algorithm, NC_DD, which disregards the criticalness of tasks when scheduling and also does not share the load across nodes. Hence, at low loads, this algorithm fails to take advantage of other free nodes in the system when it cannot guarantee a task locally. At high loads it does not maximize the collective WGR by selecting only the higher criticalness tasks when the load is such that not all tasks can be guaranteed.

The sharpest decline of WGR with an increase in load is shown by the DDLN algorithm which incorporates the unconditional guarantee. As the load increases, this algorithm does *not* adapt to the changing system conditions: it does not maximize the WGR by biasing toward the more critical tasks, instead, it persists in guaranteeing tasks based solely on their deadlines.

The fact that criticalness becomes more important than deadline as the criterion for determining the schedulability of a task at *high* loads is further corroborated by the high performance of NC_CR at high loads. NC_CR suffers from a lack of load sharing at low loads. However, despite its non-cooperative nature, its performance is very close to that of ALG1 and ALG2 at high loads, indicating that its policy of giving preference to high criticalness tasks pays off under high load conditions.

Among the cooperative algorithms, ALG1 and ALG2 give the best performance. At low loads, the algorithms that share load, ALG1, ALG2 and DDLN outperform the non-cooperative algorithms, NC_DD and NC_CR. This relative performance can be attributed to the fact that the non-cooperative algorithms do not take advantage of the free processor time at other more lightly loaded nodes in the system when the load at the local node exceeds the node's capacity to service tasks.

ALG1 and ALG2 perform well above all the (cooperative) baseline algorithms across the range of loads because of their adaptiveness and sensitivity to changing system conditions. At low loads, ALG1 and ALG2 maintain a high WGR by sharing load by reallocating tasks that cannot be guaranteed locally to other more lightly loaded nodes. And, at high loads, they are biased towards tasks with high criticalness, thus improving the overall WGR. This results in superior performance across the range of loads.

We also find that the values of WGR for ALG1 and ALG2 are very similar under most light loads. There is, however, a small improvement (of about 2%) in ALG1's WGR over ALG2's at a system load of 15. Under the system conditions considered, our simpler deadline-criticalness based algorithm, ALG2, generally performs just as well as the more complex ALG1. This might be attributed to the fact that there is an underlying trade-off

between maximizing WGR and the (time) cost of running the algorithm.

Comparing the distributed and non-cooperative deadline based algorithms, DDLN and NC_DD, we note that

- both the deadline based algorithms have worse performance than our deadline-criticalness combination algorithms.
- at low loads, DDLN outperforms NC_DD by 17.6% because of load sharing; however, the performance of the non-cooparative, NC_DD, declines slower than that of the DDLN.

From a comparison of the upper bound C_CRDD and the non-cooperative, NC_CR, we observe that

- the centralized C_CRDD is the best algorithm (in terms of WGR), giving far better results at the high loads than any other algorithm
- however, except for very low loads, the number of tasks it guarantees are much lower than our algorithms since it maximizes WGR at the expense of meeting the deadlines of more tasks. For example, at a load of R = 15, its WGR value is 89.07%, whereas its GR value is only 36% (cf. WGR of 68.85% and a GR of 53.87% for ALG1 under the same conditions).
- NC CR loses more than 20% of the tasks even under very low loads. Since it is also tailored to maximize the number of critical tasks, its WGR value is very close to ours at high loads; at $R = 15$, it is 65.84% and that of ALG2 is 66.80%.

Figure 2 shows the performance of the various algorithms as average laxity is varied from 1.0 to 10.0. Here we see that over all laxities, ALG1 and ALG2 perform better than all the baselines (except the ideal) and perform similarly to each other.

In summary, the main conclusions are:

- Our algorithms outperform all the other baseline algorithms (with the exception of the upper bound given by C_CRDD) for both the metrics, WGR and GR, under the range of system loads considered.
- Despite their different degrees of complexity, both ALG1 and ALG2, yield comparable performance.
- Our algorithms combine the advantages of both the criticalness and deadline based schemes.
- The simple deadline based algorithm, DDLN, is a sufficiently good heuristic under low overload situations. However, its performance deteriorates very quickly as the load is further increased.

Note that we can better understand the mechanics underlying our algorithms versus the often-studied deadline based scheme if we separate the notions of *schedulability* and *serviceability* as applied to *guarantee*. The schedulability of a task depends on the conditions in the system when the task arrives at the node; the serviceability is a function of how the conditions in the system change during the life-time of the scheduled but not-yet-executed task. (If we allow tasks to be removed from the guaranteed list, then serviceability would depend on how conditions change between the time when the task is scheduled and when it *completed*). In the case of an *uncondional guarantee*, a task's schedulability is the same as its serviceability. In the case of a *conditional guarantee*, a task might be schedulable, at first but become unserviceable as conditions in the system change.

6.2 System Load Distribution

We now study the effects of load imbalance across a range of system loads, with the laxity of a task again held constant at 3. Due to space constraints, we will only consider in detail the system performance under two loads of $R = 5$ and at $R = 15$, respectively, for three values of b = 0.3, 0.5 and 1.0. We then study the breakdown of *all tasks guaranteed* into those that are generated locally and those that arrive from remote nodes as a result of load sharing activities.

In Figures 3 and 4, we plot WGR against the balance factor, b, for $R = 5$ $R = 15$, respectively. In Figures 5 and 6, for ALG1 we plot the GR & WGR, with respect to only the *locally generated* tasks, against the balance factor, b for loads 2.5, 5, 10 and 15. That is, in Figure 5 we plot the ratio of the number of *local* tasks guaranteed to the number locally generated. In Figure 6, we plot the *weighted* ratios of the same quantities as found in Figure 5.

Observations and Discussion

Our algorithms perform better than the non-cooperative criticalness based baseline, NC_CR, across the range of balance conditions in terms of the percentage of tasks guaranteed (GR) although the weighted values (WGR) for ALG1 & ALG2, and NC CR are comparable at high loads (since the latter schedules tasks solely on the basis of their criticalness level).

The NC_DD also yields much poorer performance than ALG1 and ALG2 across the range of balance factors, improving slightly under conditions of high balance of load across the nodes. Hence, the generally poor performance of the non-cooperative algorithms points to the need for load distribution in any distributed system where the system load might be unevenly distributed.

At low loads, DDLN and our algorithms do equally well across all balance conditions. At high loads, our algorithms are substantially superior to this baseline. Further, we note that the WGR values for DDLN decrease when b = 1.0.

We now look at the characteristics of the guaranteed tasks for ALG1. Specifically, we compare the percentages of local and remote tasks that make up all the tasks that are guaranteed. From Figures 5 and 6, we observe that as the balance increases, for all loads, more of the tasks guaranteed are local because of the high arrival rate of local tasks. For instance, at a load of 10, with laxity 3, a breakdown of all tasks *guaranteed* into local and remote reveals that as b is increased from 0.3 to 1.0, (71.66 - 49.64 = 22.02%) *more* (guaranteed) tasks are local:

b = 0.3		b = 1.0	
% Local	% Remote	% Local	% Remote
49.64	50.36	71.66	28.34

Furthermore, the corresponding WGR values show the collective worth of the *remote* tasks accepted is much less (15.07% at b=1.0 vs. 23.20% at b=0.3) than that of the local ones:

b = 0.3		b = 1.0	
% Local	% Remote	% Local	% Remote
76.80	23.20	84.93	15.07

When we compare the performance of non-cooperative algorithms with that of distributed algorithms that incorporate load sharing, we find that load sharing is a desirable feature of a scheduling algorithm in a distributed system environment. The need for load sharing is especially felt under low load situations. In these situations, tasks that are rejected from a busy site could be reallocated to the lightly loaded alternative nodes which exist in the system. Under high system load situations however, all nodes have an abundance of task arrivals and alternative sites for rejected tasks are not readily available.

The performance of our algorithms remains constant across the two extreme conditions of total balance and high imbalance under low loads. Even at high loads, the percentage of tasks guaranteed is constant across the extremes of load balance. However, in this heavy load case, there is an improvement in WGR values with increasing balance due to the greater availablity of (local) tasks from which to maximize WGR.

In summary, there is a general consistency in performance of ALG1 and ALG2 across a wide range of system balances. Due to their load sharing policy, these algorithms take advantage of lightly loaded nodes in conditions of uneven load across the system. Conversely, in conditions of high load *evenly* distributed across all nodes, the algorithms reduce the degree of load sharing and instead, maximize the percentage of local tasks guaranteed. This concentration on the local tasks leads to fewer tasks being transferred, thus reducing the amount of processing each node carries out in order to determine the schedulablity of remote tasks as a result of bidding.

7 Conclusions

We presented two algorithms that explicitly account for deadlines and criticalness. Under the range of system conditions studied, the algorithms outperformed all the cooperative baseline algorithms. We noted that ALG2 and ALG1 have the same performance for almost all system conditions and task parameters; the difference in WGR values for the two new algorithms is within 2% of each other.

From the results presented here and in [Biya, 88], we observed that at low loads deadline based algorithms tend to perform better than criticalness based algorithms. At high loads, the situation is reversed and criticalness based algorithms outperform deadline based algorithms. Also, we show that the algorithms, ALG1 and ALG2, outperform the deadline and criticalness based baselines. Hence, our proposed algorithms combine the advantages of both the deadline and criticalness based algorithms.

The concept of 'guarantee' as *conditional* rather than *absolute* is found to be superior in terms of adaptiveness and responsiveness to changing system conditions. Separating the notion of the *schedulability* of a task from its *serviceability* is very important when considering conditional guarantee. It is this distinction that leads to a better adaptation to varying system conditions. A scheduled task may become unserviceable as a result of changes in the system. Examples of these changes could be an increased system load, a burst or influx of tasks with a particular combination of attributes (such as high criticalness and tight deadlines) or a high imbalance in system load across the various nodes.

ALG1 and ALG2 outperform even the ideal, zero-cost, non-cooperative algorithms of NC_CR and NC_DD, under almost all system conditions studied. It is shown that load-sharing is especially desirable under *low* load situations when the system is unbalanced. Under such load conditions, tasks that are not schedulable at a busy node could be reallocated to an alternative, lightly-loaded site instead of being discarded. However, in conditions of heavy load, few such lighly-loaded alternative sites exist since all nodes have an abundance of tasks. Hence, alternative sites might not be readily available for rejected tasks when the system load is high.

References

[Biya, 88] Biyabani, S. R., "The Integration of Deadlines and Criticalness in Hard Real-Time Scheduling", *Master's Thesis*, Dept. of E. C. E., University of Massachusetts, 1988.

[CL, 85] Chang, H-Y. and Livny, M., "Priority in Distributed Systems", *Proceedings of 1985 IEEE Real-Time Systems Symposium*, pp. 123-130.

[Cheng, 87] Cheng, S-C, J. Stankovic, and K. Ramamritham, "Scheduling Algorithms for Hard Real-Time Systems", *Real-Time Systems Newsletter*, Vol. 3, No. 2, Summer 1987.

[Chu, 83] Chu, W. W., "Task Response Time Model and its Applications for Real-Time Distributed Processing Systems", *Proceedings of 1984 IEEE Real-Time Systems Symposium*, pp. 225-236.

[GJ, 75] Garey, M. R. and Johnson, D. S., "Complexity results for Multiprocessor Scheduling under Resource Constraints", *Society for Industrial and Applied Mathematics Journal of Computing*, 4, 1975.

[GJ, 76] Garey, M. R. and Johnson, D. S., "Scheduling Tasks with Nonuniform Deadlines on two Processors", *Journal of the Association for Computing Machinery*, Vol. 23, No. 3, Jul. 1976.

[JLT, 85] Jensen, E. D., Locke, C. D. and Tokuda, H., "A Time-Driven Scheduling Model For Real-Time Operating Systems", *Proceedings of 1985 IEEE Real-Time Systems Symposium*, pp. 112-122.

[Lein, 80] Leinbaugh D., "Guaranteed Response Times in a Hard Real-Time Environment", *IEEE Transactions on Software Engineering*, Vol. SE-6, Jan. 1980.

[LY, 82] Leinbaugh D., and Yamini, M., "Guaranteed Response Times in a Distributed Hard Real-Time Environment", *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1982.

[LL, 73] Liu, C. L. and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, Jan. 1973, pp. 46-61.

[Mok, 83] Mok, A., "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", *Ph. D. Dissertation, Massachusetts Institute of Technology*, 1983.

[MD, 78] Mok, A., and Dertouzos, M., "Multiprocessor Scheduling in a Hard Real-Time Environment", *Proc. of the Seventh Texas Conference on Computing Systems*, Nov. 1978.

[RS, 84] Ramamritham, K. and Stankovic, J. A., "Dynamic Task Scheduling in Distributed Real-Time Systems", *IEEE Software*, Vol. 1, No. 3, Jul. 1984, pp. 65-75.

[SLR, 86] Sha, L., Lehoczky, J., and Rajkumar, R., "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling", *Technical Report*, Dept. of Computer Science, Carnegie-Mellon University, 1986.

[Stan, 85] Stankovic, J. A., "Stability and Distributed Scheduling Algorithms", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10, Oct. 1985, pp. 1141-1152.

[SRC, 85a] Stankovic, J. A., K. Ramamritham, S. Cheng, "Evaluation of a Bidding Algorithm For Hard Real-Time Distributed Systems," *IEEE Transactions on Computers*, Vol. C-31, No. 12, pp. 1130-1143, Dec. 1986.

[Stan,87] Stankovic, J. and K. Ramamritham, "The Design of the Spring Kernel," *Proc. Real-Time Systems Symposium*, Dec. 1987.

[ZRS, 87] Zhao,W., K. Ramamritham. J. A. Stankovic, "Scheduling Tasks With Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 5, May 1987.

[ZRS, 87a] Zhao,W., K. Ramamritham. J. A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints in a Hard Real-Time System," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987.

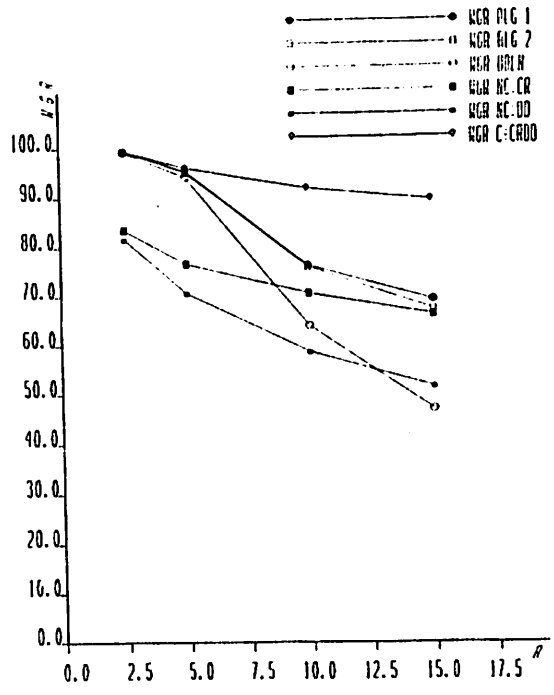


Figure 1 WGR vs System Load: b = 0.5, Laxity = 4

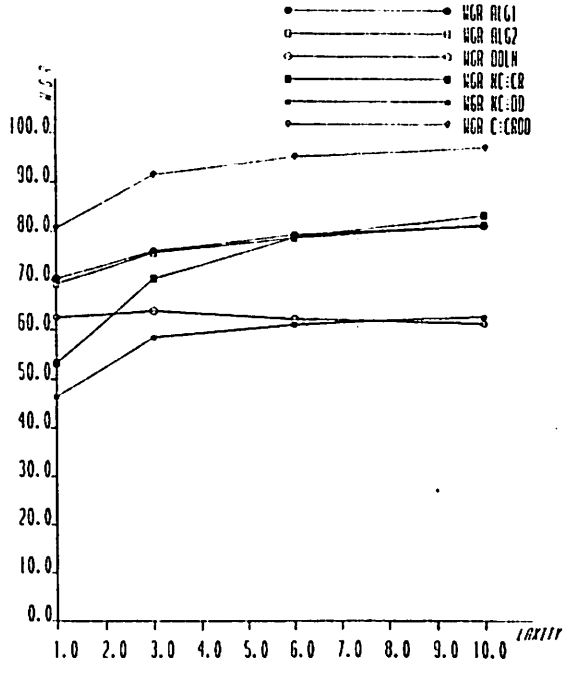


Figure 2 WGR vs. Laxity: R = 10, b = 0.5

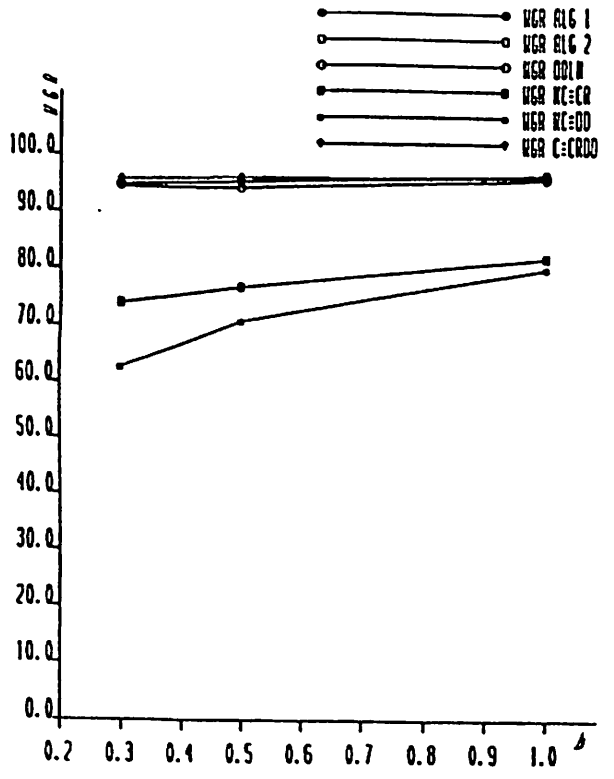


Figure 3 WGR vs. Balance Factor: R = 5.0, Laxity = 3

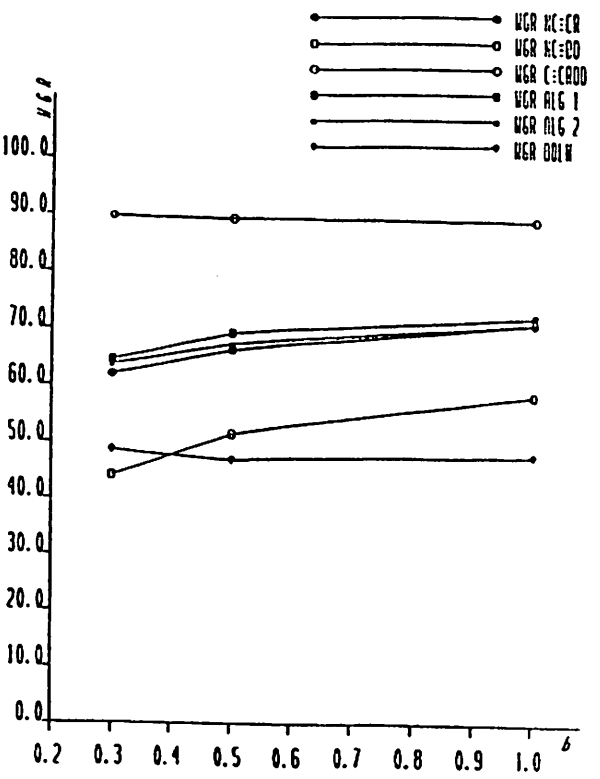
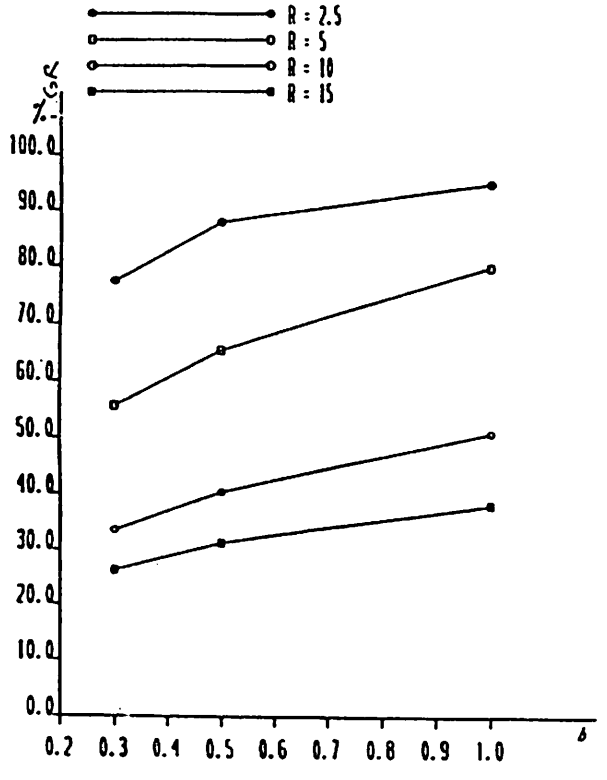


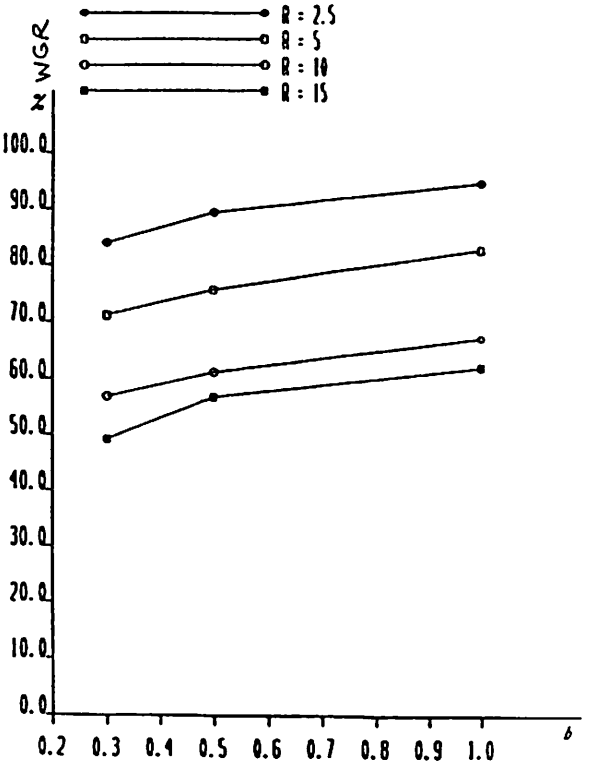
Figure 4 WGR vs. Balance Factor: R = 15.0, Laxity = 3



(Relative to GR)

Laxity = 3

Figure 5 ALG1: Percentage of tasks guaranteed locally



(Relative to WGR)

Laxity = 3

Figure 6 ALG1: Weighted Percentage of tasks guaranteed locally