Real-Time Scheduling of Multiple Segment Tasks

Kam Sang Ho, James H. Rice, Jaideep Srivastava
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

ABSTRACT

This paper studies the problem of on-line non-preemptive scheduling of multiple segment real-time tasks. Task segments alternate between using CPU and I/O resources. A task model is proposed which encompasses a wider class of tasks than models proposed earlier. Total slack in a task is modeled as a new kind of virtual resource which is spent while waiting in queues for service by physical resources. Instead of developing new scheduling algorithms, we develop a class of slack distribution policies which use varying degrees of information about task structure and device utilization to judiciously budget task slack. Several slack distribution policies are given, and one is shown to improve the performance of all scheduling algorithms studied. Two key observations are: (1) slack distribution is helpful beyond a certain threshold of task arrival rate, and (2) algorithms which normally perform poorly are helped to a greater degree by slack distribution. A study of various slack distribution policies for a constant value function reveals that all of them favor tasks with a large number of small segments to tasks with a small number of large segments. Finally it is shown that the Moore ordering algorithm, which has been reported to be optimal for single segment tasks (for the miss ratio cost measure) is not optimal for multiple segment tasks. All the observations have been verified by a comprehensive, distribution-driven simulation.

1. Introduction

A real-time system is defined as a system in which the correctness of a computation depends not only on its logical result, but also on the time at which the result is produced [DASA85]. For example, processing data from a sensor might need to occur in the microsecond or millisecond range, while consistent updates of replicated files might need to occur in the seconds range, and the movement of the material in an automated factory might have deadlines in the range of minutes or hours [STAN89].

Tasks in real-time systems have deadlines that must be met; failing to meet these deadlines could result in serious penalties. This makes the resource scheduler a key component of the system because it has the responsibility to minimize these penalties. One way of classifying resource scheduling algorithms is preemptive versus non-preemptive. In preemptive scheduling, a task in execution can be replaced by a more urgent task if the preempted task can be later finished without missing its deadline. Such replacement is not allowed in non-preemptive scheduling; a task runs until it blocks waiting on another resource or until its completion. Both preemptive and non-preemptive CPU scheduling algorithms have been studied extensively in real-time systems [LU73], [SHA86], [WOOD87], [ZHAOA79], [ZHAO87a], [ZHAO87b].

Another classification of scheduling algorithms is on-line (static) versus off-line (dynamic). Off-line scheduling is used for systems with all scheduling information known in advance. On the other hand, on-line scheduling can be used when information needed for scheduling is not known in advance. An off-line scheduler can produce a good schedule prior to execution because all information needed for scheduling is available beforehand. For real-time systems, off-line resource scheduling is possible for systems in which the task system has known structure [MO85] and, hence, all scheduling information required is known in advance. On-line scheduling is the only choice when scheduling information is not available beforehand. An example is a system with periodic tasks and random arrival-vales, where no apriori information on task arrival is available.

Garey, et al [GARE77] have shown that constructing an optimal off-line non-preemptive schedule for a single resource, with arbitrary arrival, computation, and laxity requirements is NP-complete. Hence, most of the later work has focused on developing heuristics that yield near-optimal schedules. On-line non-preemptive scheduling is NP-complete [GARE77], and a heuristic solution is the only realistic method for scheduling. A number of heuristic on-line non-preemptive scheduling algorithms have been proposed for real-time tasks [LEIN80], [MOK85], [MOK85a], [RAM84b], [WOOD87], [ZHAOA79], [ZHAO87a], [ZHAO87b].

All previous work, with the exception of Sha, et al [SHA86], has considered tasks requiring only the CPU resource. Sha, et al [SHA86] have considered I/O as another physical resource required by a task. In their model a task has a fixed number of segments (i.e. either 2 or 3), each segment corresponding to a physical resource requirement. For example, a task first makes a request to the CPU, then the I/O device, and finally the CPU again.

In this paper, we consider a model which includes a wider class of tasks. In our model a task can have multiple segments, alternating between CPU and I/O. Our aim is to model tasks which may access data from a secondary storage device. The length of each segment is a stochastic quantity, and the number of segments is variable. Sha, et al's model [SHA86] is a special case of our model where the number of segments as well as the size of each segment is fixed.

The slack time of a task is the total time it can spend waiting in various queues. A multiple segment task, requiring access to multiple physical resources, may need to wait in different queues. This gives rise to the issue of slack distribution, i.e. the maximum amount of time that can be spent waiting in the queue of a physical resource needed to process a particular segment. Sha, et al [SHA86] have proposed initially allocating all the slack to the first segment. After the execution of the segment, any remaining slack is given to the second segment, and so on. Such a policy is biased towards earlier segments and is likely to result in a very high miss ratio (the ratio of the number of tasks missing their deadline to the total number of tasks), especially if later segments require heavily utilized physical resources. Our approach considers slack itself an expendable virtual resource to be distributed among task segments. We use information about (a) task structure and (b) device utilization to distribute slack in a more effective manner. We have developed a class of slack distribution policies that can be used with existing real-time scheduling algorithms to improve their performance. In a comparative study on non-preemptive on-line real-time scheduling algorithms, Woodside [WOOD87] showed that Earliest Due Date (EDD), Latest Start Time (LST), and Moore Ordering (MM) show good behavior for single segment tasks.

Combining Sha, et al's [SHA86] idea of deadline propagation1 for multiple segment tasks, and our idea of slack distribution, we develop a new class of algorithms. An example is the Propagated Earliest Due Date with Task information based slack distribution (P-EDD/T) algorithm. We carried out extensive simulations which show that these algorithms

---

1 Task deadline is the time by which the whole task must complete. For a multiple segments task, with successive segments requiring different resources, each segment is the minimal scheduling unit. If the processing time of each segment is known, the task deadline can be propagated forward to determine the deadline by which each segment must complete.
perform better than propagated algorithms without slack distribution, especially at high system load. This confirms the fact that slack distribution is a worthwhile idea. Another contribution is that we study the effect of using different definitions of a task value function for a performance metric.

The remainder of this paper is organized as follows: Section 2 describes the task model we adopt in this study. Section 3 discusses the existing real-time scheduling algorithms while Section 4 introduces our slack distribution policies. Section 5 describes our simulation model. Section 6 presents the results of simulation studies and Section 7 provides a summary.

2. Task Model

In a real-time environment, tasks can be characterized by the following three properties: the nature of arrival, structure, and value.

2.1. Nature of Task Arrival

The arrival characteristics of a class of tasks can be described as periodic or aperiodic. The inter-arrival time of periodic tasks is fixed and is known in advance. For example, a sensor detects the temperature of a nuclear reactor at fixed intervals and sends the data to the system for processing. The inter-arrival time of aperiodic tasks, on the other hand, can be either known or unknown. In our study, we model a system that has aperiodic tasks with stochastic inter-arrival times, making it impossible to predict future arrivals. An example is arrival of jobs at a service station.

In periodic systems, complete knowledge about arrival of tasks is available in advance and off-line static scheduling can be carried out. Such an approach has been taken by the SARTOR system [MOK87]. In some systems, off-line scheduling is not possible because complete knowledge about arrival of tasks is not available apriori. On-line scheduling must be used in this case. In this study, since complete apriori knowledge of tasks entering the system is not available, off-line scheduling is not possible.

2.2. Task Structure

Resources required by a task can include CPU, disk, and other hardware devices. Of all the resources, CPU and disk are the most common. In this study, we are interested in modeling systems where part of the data resides in a file stored on a secondary storage device such as a disk. Since tasks in our model require multiple CPU and I/O bursts, a task structure with multiple segments is reasonable.

Tasks must demand the CPU at least once, sometimes followed by an I/O burst for data file access, and then another CPU burst. This CPU-I/O cycle can repeat any number of times, finishing with a final CPU burst. A task’s physical resource requirements can be viewed as consisting of multiple segments, alternating between CPU and I/O. The first segment of a task is always a CPU burst (C), the second segment is an I/O burst (I), the third is a CPU burst, and so on. We model task execution only and assume that a core image of the task is present, i.e. context switching is included in scheduling overhead. Thus, the task structure can be expressed by the regular expression C(*I*C)*. The number of segments and the length of each segment is a stochastic quantity which becomes known on arrival. Other known characteristics of the task are its arrival time, $T_a$, release time, $T_r$, deadline, $T_d$, and the current time, $t$.

Arrival time is a time stamp generated upon a task’s arrival. Task deadline, $T_d$, is the latest time by which the task must complete. Release time of a task is the earliest time at which it can begin execution, which in general may be greater than its arrival time. We assume a task is ready for execution as soon as it arrives, i.e. $T_r = T_a$. Other scheduling information can be computed by using the current system time, task deadline, and the C(*I*C)* information. If $t$ is the current system time, the following attributes can be computed:

- $T_a$: total processing time left is the total time required to execute the remaining C’s and I’s;
- $T_r$: remaining slack at current system time, i.e., $T_d - T_r - t$;
- $T_d$: latest start time, i.e., $T_d - T_r$.

2.3. Task Value

The performance of a number of real-time CPU scheduling algorithms has been compared by Jensen, et al. [JENS85]. The concept of a value function is introduced, which is a measure of the value obtained by completing a task as a function of time. This provides a uniform framework for handling both hard real-time and soft real-time tasks. For hard real-time tasks, the value function drops to zero or negative if the task becomes tardy. The value function of soft real-time tasks drops gradually after its deadline as a function of time. We use a value function based on the total processing time of a task, which is known upon task arrival. In addition, our study focuses on hard real-time tasks.

Rejection ratio (or Miss ratio), i.e. fraction of submitted tasks missing their deadline, has been used by many researchers as a parameter to measure the system performance. It is adequate for systems with homogeneous tasks, or tasks with the same level of importance even though differing in their lengths, $L$, which is the sum of the lengths of the C and I segments. We model this by considering the following constant value function, $V(L)$:

- Constant value function: $V(L) = k$;

For tasks with different sizes and/or different levels of importance, rejection ratio may not be a good performance measure. From a resource utilization viewpoint, task value should be a function of task length, since this is an indicator of the amount of physical resources used by it. A task with large physical resource requirements in general should have a high value. We model this by considering the following linear cost function:

- Linear value function: $V(L) = L$;

The constant function assigns equal value to all the tasks, i.e. it assumes all tasks have the same importance. The linear function assigns higher values to larger sized tasks. System value, $SV$, is the sum of values for tasks that complete before their deadlines. We believe the linear function is a better measure of system performance since it captures the characteristics of a task more accurately. Let,

$$SV = \sum_{T_i} V(\text{Value}(T_i))$$

For a constant value function, the task rejection (miss) ratio, $RR$, has a simple relationship to the system value ratio, i.e., $RR = 1 - SVR$.

3. Real-Time Scheduling Algorithms

Various studies have compared the performance of many on-line, non-preemptive real-time scheduling algorithms. The First In First Served (FIFS) algorithm does not consider the real-time nature of tasks, but is often used as a base reference. Many algorithms have been devised which consider the real-time nature of tasks. However, as shown by Woodside [WODD87], the best performance is by EDD and MM (which is a more elaborate version of EDD) as shown in Fig. 1. For preemptive scheduling, Dertouzos [DERT74] has shown that LST is as good as EDD. Since these are the best algorithms for single segment tasks, we decided to concentrate on those algorithms for our study. For multiple segment tasks with different resources required for successive segments, consider propagating deadline (i.e. deadline of the current segment) for scheduling, can be used, as shown by Sha, et al [SHA86]. They devised the propagated version of EDD, which we call P-EDD. We extended the idea to LST and MM to obtain the P_LST and P_MM algorithms.

Scheduling algorithms which do not introduce intentional delay can be thought of as maintaining a priority queue of tasks, with the priority being the scheduling criterion. FIFS inserts tasks into the priority queue based on task entry time. EDD and MM insert tasks into the queue based on deadline time. LST inserts tasks based on task’s earliest start time. Similarly, P-EDD and P-MM insert tasks based on task segment deadline, while P_LST inserts tasks based on task segment latest start time. The following is a brief description of multiple segment scheduling algorithms that we have studied:

- FIFS: Task segments are serviced in the order of arrival at a device. This algorithm does not consider the real-time nature of tasks, and has been studied for comparison with other algorithms.
- EDD: The task deadline is the deadline of its final segment. Task segments are serviced in the order of earliest task deadline.
LST: For each task, the latest start time, i.e. the difference between its task deadline and its processing time, \( T_0 \), is calculated. Task segments are executed in the order of increasing latest start time.

MM: Tasks are ordered by earliest task deadline. On the completion of a segment’s execution, task segments in the queue are examined in order, and when a segment is discovered to be infeasible, the one with the largest computation requirement from the set of already guaranteed (scheduled but not yet executed) tasks is removed. This is done until all tasks have been examined.

P-EDD: Given the task’s ultimate deadline, \( T_0 \), the number of segments \( n \), and the length, \( L_i \), of segment, \( S_i \), the propagated deadline, \( PD_i \), for the \( i^{th} \) segment, \( S_i \), is:
\[
PD_i = T_0 - \sum_{j=1}^{i} L_j, \quad 1 \leq i \leq n
\]

Task segments are serviced in the order of earliest propagated deadline (i.e. segment deadline) instead of earliest task deadline, i.e. the segment with the most urgent propagated deadline gets serviced first. Tasks whose propagated deadline have expired are removed from the system.

P-LST: This algorithm is similar to the P-EDD algorithm, except that propagated latest start times are calculated. Propagated latest start time, \( PL_i \), is calculated by:
\[
PL_i = PD_i - L_i, \quad 1 \leq i \leq n
\]

Task segments are serviced in the order of earliest latest start times of the current segments, i.e. the task segment with the most urgent latest start time is serviced first. A task whose propagated latest start time is missed is removed from the system.

P-MM: Similar to P-EDD, except that the MM type feasibility analysis is done for all segments in the queue.

Note that all these policies, i.e. EDD, EDD, LST, MM, etc. differ only in task ordering. Hence, each of these can be considered as a priority-driven policy. Real-time scheduling algorithms that perform well use more information about the real-time nature of tasks [SHA86], [WOOD87]. Figure 1 shows results of the Woodside study. We ran simulations using single segment tasks with Poisson arrival-rates and exponentially distributed processing times, as assumed by Woodside, to verify the study.

4. Slack Distribution

The latest start time (LST) of a task is the difference between its task deadline and the current system time, i.e. it is the amount of time that a task can afford to wait in queues before getting serviced. A multiple segment task is queued for service many times. Time spent in each queue can be considered as using up the task’s slack. For a task to complete successfully, the sum of its waiting times in all queues must be no more than its total slack. Slack distribution is the mechanism to distribute the slack among the segments of a task.

Thus, only a portion of the slack is allocated to the task segment to be executed next. If the segment cannot begin service within its slack, it is removed from the system regardless of any remaining overall slack. Slack distribution for all segments becomes an important issue. The distribution policy should be designed to utilize slack in the most effective manner. Almost no previous work considered slack distribution because tasks had only one segment, i.e. tasks required service only from one resource.

Sha, et al [SHA86] considered tasks with more than one segment. Their approach was to assign all the available slack to the segment to be executed next. This approach is biased towards the initial segments, providing little, if any, leeway to later ones. Slack distribution is a dimension distinct from task ordering. It is applicable to any real-time scheduling algorithm dealing with multiple segment tasks. Our study includes the algorithms EDD, LST, MM, P-EDD, P-LST, and P-MM. Our study shows how the performance of these scheduling algorithms is improved by incorporating slack distribution policies.

4.1. Factors Determining Slack Distribution

Slack can be distributed among task segments based on various criteria. We consider a class of criteria which take into account varying amounts of information about task structure and device utilization. Task structure information includes the number of remaining task segments, the length of each segment, and the type of resource required next (CPU or I/O). The strategy for using task information is to give task segments with large processing requirements more slack than smaller task segments. Device utilization information includes overall CPU utilization and overall I/O utilization. The strategy for using system utilization is to allocate more slack to task segments requiring service from busy devices. These segments require more slack because they must wait longer in their device queues.

4.2. Slack Distribution Policies

We classify slack distribution policies according to the amount and type of information they use. These range from using no information to using both the task structure and device utilization information. Symbols used in the formulas are defined in Table 1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>Number of task segments</td>
</tr>
<tr>
<td>( S_i )</td>
<td>The ( i^{th} ) segment of a task</td>
</tr>
<tr>
<td>( L_i )</td>
<td>Length of segment ( S_i ), a random variable with exponential distribution</td>
</tr>
<tr>
<td>SlackAlloc</td>
<td>Slack allocated to the current task segment</td>
</tr>
<tr>
<td>RemSlack</td>
<td>Remaining slack, ( T_0 - PL_i - \sum_{j=1}^{i-1} L_j ) processing time</td>
</tr>
<tr>
<td>( PD_i )</td>
<td>Propagated deadline for segment ( S_i )</td>
</tr>
<tr>
<td>( PL_i )</td>
<td>Propagated latest start time for segment ( S_i )</td>
</tr>
<tr>
<td>( U_i )</td>
<td>Cumulative CPU utilization</td>
</tr>
<tr>
<td>( U_d )</td>
<td>Cumulative Disk utilization</td>
</tr>
</tbody>
</table>

4.2.1. No Information Based

SLR: This is the slack distribution policy proposed by Sha, et al in their propagated earliest due date scheduling algorithm [SHA86]. The allocated slack for the current segment is, SlackAlloc = RemSlack. This policy is increasingly unfair to later task segments because they receive slack only if it is unused by the previous segments. The first task segment is awarded the total task slack. If it happens to use up all of it, the remaining segments must be scheduled with no queuing delays, which can be quite difficult.

4.2.2. Task Information Based

EQUAL: The current task segment, \( S_i \), is awarded a fraction of the slack based on the number of remaining task segments. Thus, the allocated slack for the current task segment is:
\[
\text{SlackAlloc} = \frac{\text{RemSlack}}{n - i + 1} \text{RemSlack}
\]

TASK: The current task segment, \( S_i \), is awarded a fraction of the remaining slack based on its length. More slack is awarded to task segments having larger processing (CPU or I/O) requirements. Allocated slack is:
\[
\text{SlackAlloc} = \frac{\sum_{j=1}^{i} L_j}{\sum_{j=1}^{n} L_j} \text{RemSlack}
\]

4.2.3. System Information Based

SYSTEM: The current task segment is awarded a fraction of the remaining slack based on its resource requirement type (CPU or I/O) and the device utilizations. More slack is awarded to segments using busy devices. Allocated slack is:
\[
\text{SlackAlloc} = \frac{U_i}{U_i + U_d} \text{RemSlack} \text{ for CPU segment;}
\]
\[
\text{SlackAlloc} = \frac{U_d}{U_i + U_d} \text{RemSlack} \text{ for I/O segment;}
\]
4.2.4. Hybrid Information Based

HYBRID1: Slack is distributed based on the current segment type (CPU or I/O) and the number of remaining segments. A mixture of task and system attributes are used. Slack allocated to the current segment, $S_i$, is:

$$\text{SlackAlloc} = \text{RemSlack} \left( \frac{1}{n-i} + \frac{1}{n-i+1} \right),$$

for CPU segment;

$$\text{SlackAlloc} = \text{RemSlack} \left( \frac{1}{n-i} \right),$$

for I/O segment;

The denominator is the sum of $n-i+1$ device utilizations which correspond to the $n-i+1$ remaining task segments. Thus, the current segment is awarded slack based on its type, the device utilizations, and the number of remaining task segments.

HYBRID2: Slack is distributed based on another mixture of task and system attributes. Allocated slack is:

$$\text{SlackAlloc} = \text{RemSlack} \left( \frac{1}{n-i} + \frac{1}{n-i+1} \right),$$

for CPU segment;

$$\text{SlackAlloc} = \text{RemSlack} \left( \frac{1}{n-i+1} \right),$$

for I/O segment;

Thus, the current task segment is awarded a fraction of the remaining slack based on its computation time (CPU) or I/O, the length of remaining task segments, and the device utilizations. All of the following slack distribution policies can be used in conjunction with each of the multiple segment scheduling algorithms, namely EDD, LST, MM, propagated earliest due date (P_EDD), propagated latest start time (P_LST), and propagated Moore ordering (P_MM). Each policy calculates the amount of slack which is allocated to the current segment.

5. Simulation Model

Our approach has been evaluated by means of a comprehensive, distribution driven simulation. As shown in Figure 2, the simulation model consists of a CPU, an I/O device, an CPU scheduler, and an I/O scheduler. A CPU queue is attached to the CPU and an I/O queue is attached to the I/O device. The queues are used to store the tasks that are scheduled and are ready for service. In order to examine the performance of the scheduling algorithms alone, no scheduling overheads are considered.

5.1. Processes and Their Interaction

There are five processes running "in parallel" in the simulation. These processes are:

- Task generator: Generates new tasks with an exponentially distributed inter-arrival time. The length of each segment is exponentially distributed on a random number generator.
- CPU scheduler: Inserts tasks into the CPU queue based on the appropriate scheduling criteria.
- CPU: Removes the task from the CPU queue and services it at its deadline. Otherwise discards it.
- I/O scheduler: Inserts tasks into the I/O queue based on the appropriate scheduling criteria.
- I/O device: Removes the task from the I/O queue and services it at its deadline. Otherwise discards it.

When a task is generated, the CPU scheduler inserts the task into the CPU queue based on the appropriate scheduling criteria. On completing its current task segment, the CPU retrieves the task at the head of the CPU queue. If the segment's deadline can be met, it is executed; otherwise the whole task is discarded. Upon completion, the CPU examines the current task segment. If the last segment of the task has been completed before its deadline, total system value is incremented by the value of this task. The task then exits the system. Otherwise, it is sent to the I/O sub-system to complete its next segment. The I/O scheduler inserts the task in the I/O queue based on the appropriate scheduling criteria. On completing a task segment, the I/O device pulls the first task from the I/O queue and executes the task for the amount of time specified in the current segment, discarding the task if this segment's deadline has expired. On completion the task is re-routed back to the CPU. This cycle continues until the last segment of the task has been executed. Note that the processing of the segments of a task is strictly sequential.

5.2. Simulation Parameters

There are two kinds of system parameters: system and task. These simulation parameters are summarized in Table 2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of segments per task, n</td>
<td>1, 3, 5, or 9</td>
</tr>
<tr>
<td>Segments size</td>
<td>Mean 1, 3, or 10 time units, exponentially distributed</td>
</tr>
<tr>
<td>Task arrival rate</td>
<td>Mean 0.1 to 1.3 tasks/unit time, exponentially distributed</td>
</tr>
<tr>
<td>Release time</td>
<td>Same as arrival time which depends on the arrival rate</td>
</tr>
<tr>
<td>Deadline</td>
<td>Three times the expected task size, exponentially distributed</td>
</tr>
<tr>
<td>No. of tasks per run</td>
<td>13,000 tasks</td>
</tr>
</tbody>
</table>

Table 2. Simulation Parameters.

5.2.1. Task Characteristics

The number of segments in a task is uniformly distributed between 1 and 9. Since the first and the last task segments must be CPU segments, the number of segments, n, is an odd number. The length of each segment is exponentially distributed with mean 1, 5, or 10 time units. The task arrival rate is Poisson, with means ranging from 0.1 to 1.3 tasks/unit time in steps of 0.1. The task release time is the same as the arrival time of a task. Thus, a task is ready for service upon entering the system. The task deadline is the task arrival time plus an exponential distribution of three times the expected task size. The expected task size is the sum of expected CPU time, expected I/O time, and expected I/O access time requested by a task.

5.2.2. Device Characteristics

In our model there is a single application processor and a dedicated scheduling processor. Our simulation models a disk drive in a detailed manner, taking into consideration seek, latency, and transfer times. Given the request size and disk rotation speed, it is possible to calculate the transfer time. However, seek and latency depend on the starting position of the disk head, and the size of the requested data. This leads to a degree of randomness in the seek and latency which our simulation models in detail.

In our model, the disk drive has 80 cylinders and the amount of time taken to move from one cylinder to another is limited, based on the number of cylinders the head moves. The I/O drive process keeps track of the current head position. When an I/O is requested, the head is moved to the correct cylinder, chosen from a uniform random distribution, and the time taken for this head movement is the seek time. Average latency is taken to be the time for half a rotation.

6. Analysis of Simulation Results

We first compare six scheduling algorithms for multiple segment real-time tasks using no slack distribution. Then we add slack distribution policies to enhance performance.

6.1. Simulation Parameters

Based on past results, as discussed in section 3 and in Fig. 1, we decided to study MM, EDD, LST, P-MM, P_EDD, and P_LST. It is interesting to note that P_LST and LST are equivalent when no slack distribution is used. The propagated latest start time is exactly the same as the absolute latest start time when slack is not distributed among task segments. We extend the task model to include multiple segments and use non-preemptive scheduling. Then we incorporate slack distribution policies for increased performance. Our performance metric is system value and we note that performance may vary with differing definitions of task value. Simulations are run using the constant and linear task value functions described in section 2.3.

6.2. Results for Algorithms with No Slack Distribution

Figures 3 and 4 compare MM, EDD, LST, P-MM, P_EDD, and P_LST using a constant value function and linear value function, respectively. In both figures, all algorithms perform similarly until the task arrival rate rises above 0.5 tasks/unit time. After this point, the LST algo-
When the constant value function is used as the performance metric, LST is clearly the worst performing algorithm at high system load and P-EDD is marginally better than P-MM. When the linear value function is used, LST, P-EDD, and P-MM perform similarly.

6.3. EDD and P-EDD with Slack Distribution Policies

Figures 3 and 6 show the results of adding slack distribution policies to the EDD and P-EDD scheduling algorithms, respectively, using the constant value function performance metric. The most interesting feature of these figures is the crossover phenomenon. In Figure 5, the EDD algorithm without slack distribution performs better than the ones with slack distribution until the task arrival rate rises above 1.2. In Figure 6, the P-EDD algorithm without slack distribution is superior to the ones with slack distribution until the arrival rate rises above 0.7.

Also apparent in these figures is that the SYSTEM slack distribution policy consistently performs poorly. This is most probably due to the fact that the system utilization function provided by C-Sim is cumulative rather than instantaneous utilization. Thus, it does not accurately capture the system load. We expect a better indicator of the instantaneous system load, such as a moving average of the device utilizations would perform better. The TASK and HYBRID2 policies perform almost equivalently and are the best slack distribution policies in both figures. It is interesting to note that the slack distribution policies help the P-EDD algorithm more than they help the EDD algorithm.

Figures 7 and 8 show the results of adding slack distribution policies to the EDD and P-EDD scheduling algorithms, respectively, using the linear value function performance metric. Again the crossover phenomenon is apparent. In Figure 7, the EDD algorithm without slack distribution policies is superior to the ones with slack distribution policies until the arrival rate rises above 1.1. In Figure 8, P-EDD with no slack distribution is superior until the slack distribution policies contribute similarly to enhance performance. For the EDD algorithm, the slack distribution policies contribute similarly to enhance performance. However, for the P-EDD algorithm, the TASK and HYBRID policies are consistently better than the SYSTEM policy.

6.4. LST and P-LST with Slack Distribution Policies

Figures 9 and 10 show the results of adding slack distribution policies to the LST and P-LST scheduling algorithms, respectively, using the constant value function performance metric. The crossover phenomenon is even more visible in these figures. In Figure 9, LST without slack distribution is superior until the arrival rate rises above 0.8. In Figure 10, P-LST without slack distribution is superior until the arrival rate rises above 0.7. As in the EDD and P-EDD case, the SYSTEM slack distribution policy performs poorly for LST and P-LST. The TASK and HYBRID2 policies perform similarly well. Because LST and P-LST are equivalent when no slack distribution is used, the slack distribution policies help LST and P-LST similarly.

Figures 11 and 12 show the results of adding slack distribution policies to the LST and P-LST scheduling algorithms, respectively, using the linear value function performance metric. The crossover phenomenon is apparent again in both figures. In Figure 11, LST without slack distribution is superior until the arrival rate rises 0.8. In Figure 12, P-LST without slack distribution is superior until the arrival rate rises 0.7. Like Figures 9 and 10, Figures 11 and 12 are very similar. The SYSTEM slack distribution policy consistently performs poorly and the TASK and HYBRID2 policies are the best. The slack distribution policies seem to help LST and P-LST similarly independent of the performance metric.

6.5. Comparison of P-EDD to P-MM

Moore has shown that the MM non-preemptive scheduling algorithm is optimal for single segment tasks [MOOR68]. It might be expected that the P-MM scheduling algorithm with some slack distribution policy would be better than the P-EDD algorithm. However, Figure 13 shows the contrary. The HYBRID2 slack distribution policy was the best for both P-EDD and P-MM. When these two algorithms are plotted together, it is clear that P-EDD is superior. Note that the guarantee of the P-MM algorithm only guarantees that all task segments will not miss their deadlines. It does not guarantee that the tasks themselves will ultimately finish on time. In fact, the guarantee of P-MM is that a task may be killed off tasks prematurely, thus lowering the system value. This result is apparent independent of the performance metric.

6.6. Contribution of Slack Distribution Policies

Figure 14 shows how the coupling of propagation and slack distribution can improve a known poor algorithm so much that it performs better than a known good algorithm. It has been shown that EDD is superior to LST for non-preemptive scheduling of single and multiple segment tasks. However, when the P-LST algorithm is used with the HYBRID2 slack distribution policy, it performs better than the EDD algorithm alone. Thus, our study of slack distribution policies was indeed a worthwhile effort.

6.7. Additional Observations

The crossover phenomenon observed in Figures 5 through 12 suggests that the slack distribution policies are too pessimistic at low system loads, i.e., they unnecessarily kill off tasks which could have met their deadlines. However, at high system loads, the slack distribution policies are helpful because they remove potentially tardy tasks from the system early, thus freeing resources for other tasks.

7. Conclusion

Many interesting real-time scheduling problems are known to be NP-complete [GRAH79]. This has led to the design of a number of heuristic algorithms. Many researchers use the slack distribution policies as part of these algorithms. This is not surprising since it is a dimension independent of the task segment. The slack distribution policies help LST and P-LST similarly independent of the performance metric.

Woodside, et al.'s study [WOOD87] showed that EDD and MM are good algorithms for on-line non-preemptive scheduling of single segment real-time tasks. We show that their results hold for multiple segment tasks, too. According to Woodside, MM behaves uniformly better than EDD because it uses extra effort in selecting promising tasks to retain, while discarding potentially tardy ones. This does not help for multiple segment tasks because the guarantee provided by the MM algorithm, which is based on one segment at a time, is not perfect any more.

We have shown that slack distribution policies are indeed effective in improving the chances of a task meeting its deadline. This is most clearly seen in the case of the P-LST/H2 algorithm (a modified version of LST with hybrid slack distribution), which outperforms EDD, even though basic LST is much worse than EDD [WOOD87]. The improvement due to slack distribution is seen uniformly across all scheduling algorithms. This is not surprising since it is a dimension independent of task ordering, which is the dimension distinguishing various algorithms. Another observation is that the improvement provided by slack distribution policies is much better for algorithms that perform poorly otherwise. Additionally, observe that in almost all graphs there is a crossover phenomenon as a task arrival rate of approximately 0.2. It is only after this point that slack distribution leads to improvement. This is so because slack distribution leads to more pessimistic policies, i.e., they do not give as much chance to a task segment as no distribution would. Thus, at low arrival rates, they tend to kill off tasks that may actually have completed. However, at high arrival rates this pessimistic approach helps since it leads to early identification of potentially tardy tasks.

Another conclusion discussed in this paper is the sensitivity of the performance of scheduling algorithms to the metric used, i.e., the nature of the value function. In their comparative study, Woodside et al. considered miss ratio as the performance metric. This is a relevant metric for the performance metric, with respect to task length, which is satisfactory if all
tasks are similar (since it assumes that each task adds the same amount to the system value upon completion). However, if two tasks have widely different lengths, i.e. resource requirements, the value that each adds to the system should be different, too. This is intuitive since a user would expect the value received from a task to be proportional to the amount of resources invested in it. With this in mind we compared the performance of scheduling algorithms with respect to a linear, i.e. first order value function. It was observed that the performance of algorithms with the zero-th order measure was better than with the first order measure. From this we conclude that the algorithms studied prefer a large number of small tasks to a small number of big tasks. Thus, their performance for superlinear value functions is not expected to be very good.

Finally, we observed that there is a positive relationship between the amount of information used by a slack distribution policy, and the improvement caused by it. Thus, the hybrid policies are better than those using just one kind of information. Also, the improvement due to slack distribution policies based on device utilization is surprisingly marginal. We expect the improvement to be better if a moving-average type statistic is used. This is most probably because the statistics maintained by C-Sim are cumulative from the start of the simulation rather than instantaneous. We are currently working with a better instantaneous system load estimator. All the observations presented above have been verified by means of an exhaustive, distribution driven simulation.

8. References


<table>
<thead>
<tr>
<th>Figure 3: Comparison of Scheduling Algorithms without Slack Distribution, V(L) = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4: Comparison of Scheduling Algorithms without Slack Distribution, V(L) = L</td>
</tr>
<tr>
<td>Figure 5: EDD with Slack Distribution, V(L) = 1</td>
</tr>
<tr>
<td>Figure 6: P_EDD with Slack Distribution, V(L) = 1</td>
</tr>
<tr>
<td>Figure 7: EDD with Slack Distribution, V(L) = L</td>
</tr>
<tr>
<td>Figure 8: P_EDD with Slack Distribution, V(L) = L</td>
</tr>
<tr>
<td>Figure 9: LST with Slack Distribution, V(L) = 1</td>
</tr>
<tr>
<td>Figure 10: P_LST with Slack Distribution, V(L) = 1</td>
</tr>
<tr>
<td>Figure 11: LST with Slack Distribution, V(L) = L</td>
</tr>
<tr>
<td>Figure 12: P_LST with Slack Distribution, V(L) = L</td>
</tr>
<tr>
<td>Figure 13: Comparison of P_EDD and P_MM, V(L) = 1</td>
</tr>
<tr>
<td>Figure 14: Contribution of Slack Distribution Policies, V(L) = L</td>
</tr>
</tbody>
</table>