BOUNDS ON THE SCHEDULE LENGTH OF SOME HEURISTIC SCHEDULING ALGORITHMS FOR HARD REAL-TIME TASKS

F. Wang, K. Ramamritham, and J. Stankovic
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-67
June, 1990

Bounds on the Schedule Length of Some Heuristic Scheduling Algorithms for Hard Real-Time Tasks *

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

June 1990

Abstract

For algorithms that schedule hard real-time tasks, both the ability to generate feasible schedules and the quality of the generated feasible schedules, expressed in terms of the schedule length, are important performance metrics. The SPRING project uses a heuristic algorithm which integrates CPU and resource requirements for on-line, hard real-time task scheduling. It has been shown earlier, via simulation studies, to have good performance, with respect to its ability to find feasible schedules. This paper analyzes its performance with respect to the schedule length bound. Specifically, we show that its schedule length bound is equal to m, the number of processors. We identify the weakness of the heuristic algorithm which causes the worst case bound to be m and develop a new algorithm, one which attempts to keep two processors busy if possible. We analyze the schedule length bound for the new algorithm and show that it is equal to $(m+1)/2$, that is, the bound is reduced almost by half. We provide simulation results to show that the new algorithm has almost the same ability for finding feasible schedules as the original algorithm. Then we generalize this algorithm to keep at least k processors busy, if possible, and derive its schedule length bound. We also generalize the traditional list scheduling algorithm along the same lines and show the result of its analysis.

*This work is part of the Spring Project at the University of Massachusetts and is funded in part by the Office of Naval Research under contract N00014-85-K-0398 and by the National Science Foundation under grant DCR-8500332.

1 Introduction

Hard real-time systems are defined as those systems in which the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced. Examples of the hard real-time systems are command and control systems, flight control systems, space shuttle avionics systems, and robotics systems. Designing and building *dependable* and *predictable* hard real-time systems is an important research issue.

Inside a hard real-time system, there exist many time constrained activities which control the behavior and performance of the whole system. At a certain level, these activities are abstracted as *hard real-time tasks.* Generally, several tasks may share the system-level and user-level resources. This creates conflicts between the tasks when they access shared resources. *Hard real-time task scheduling* is a way to resolve the conflicts. A scheduler creates an order of execution for the tasks such that every task is completed before its deadline, receives all requested resources, and there are no conflicts in accessing the resources. Such an ordering is termed a *feasible schedule.*

One of the goals of hard real-time scheduling is to find a feasible schedule. It is a hard problem and has been shown to be NP-complete [6, 3] even for simple models. There are two ways to deal with the problem. One uses an exhaustive enumeration algorithm, such as branch and bound, or dynamic programming. The other uses approximation algorithms. Due to their large execution overheads, exhaustive enumeration algorithms may not be applicable in dynamic situations. It is an important research issue to find good approximation algorithms.

There are two important aspects to the performance of a scheduling algorithm for hard real-time tasks. The first is the *ability* of an algorithm to generate a feasible schedule. This ability is commonly characterized by the mean behavior of the algorithm, which can be measured either by a probability model or by a simulation study. For simple systems, the probability model works well. For complex systems, simulation is commonly used. The second aspect is the *quality* of the feasible schedules generated by an algorithm. Quality here is characterized by both the mean behavior and the worst case behavior of the algorithm, which are characterized by metrics such as *schedule length* and *earliness* (the distance between a task's completion time and its deadline). In this paper we deal with the worst case behavior and use the schedule length metric. Schedule length is an important characteristic of a schedule for tasks in a real-time system for the following reasons:

- For real-time task scheduling, if one algorithm generates a shorter schedule than another algorithm, it means that, in general, the first algorithm can more easily accommodate tasks with shorter deadlines than the second algorithm.

- If the schedule length is shorter, it is a more efficient schedule and it is more likely that there is more execution time remaining at the end of the schedule which can be used for executing other tasks, such as diagnostic tasks, or handling more tasks, in the case of dynamic arrivals.

We define a schedule length *bound* of one algorithm relative to an optimal algorithm in the following way. Let x be any instance of a scheduling problem, $L_A(x)$ be the schedule length of an approximate scheduling algorithm A for instance x, and $L_O(x)$ be the schedule length of an optimal scheduling algorithm O for x. If there exists a constant B, such that

$$\frac{L_A(x)}{L_O(x)} \leq B, \qquad \text{for every } x,$$

then B is called the bound for algorithm A. B is *tight* if it can be reached asymptotically. We simply use $L_A/L_O \leq B$ to represent the above inequality if there is no ambiguity.

A number of results have been published concerning the schedule length bound for non real-time task scheduling algorithms. Here we mention some important results, all of them are based on list scheduling which has a polynomial time complexity (of at most $O(n^2)$, where n is the number of tasks). More detailed information can be found in [6, 2]. Let m be the number of processors and r be the number of resource types. If $r = 1$ and all tasks have the same computation times, [7] gives a schedule length bound of $(27/10 - 24/(10 \cdot m))$. If preemptions are allowed and tasks have arbitrary computation times, the schedule length bound becomes $(3 - 3/m)$ [7]. If $r = 1$, for tasks with arbitrary computation times and precedence constraints, the schedule length bound becomes m [3]. For any m, r, and tasks with arbitrary computation times and no precedence constraints, the schedule length bound is the minimum of $(m + 1)/2$ and $r + 2 - (2r + 1)/m$ [3]. In general, list scheduling has poor ability to find a feasible schedule for hard real-time tasks because it strictly follows the earliest task's start time first policy at each scheduling decision point, thereby making deadline considerations secondary.

There are three main contributions of this paper. The first contribution is that we analyze the schedule length bound for a heuristic algorithm for hard real-time task scheduling and show that it is equal to m, which is the number of processors. The heuristic algorithm analyzed was

developed for the SPRING project [8, 9, 10, 11, 12, 13], and it is perhaps the only on-line hard real-time task scheduling algorithm which integrates CPU and resource requirements. Here we supply a theoretical basis for this heuristic algorithm. Details of the heuristic algorithm are given later. The second contribution is that we identify a weakness of the heuristic algorithm which causes the worst case schedule length bound to be m, and develop a new algorithm which attempts to keep at least two processors busy if possible. We analyze the schedule length bound for the new algorithm and show that it is equal to $(m+1)/2$, that is, the bound is reduced almost by half. The complexity of both the (new and old) algorithms is $O(n^2r)$. We provide simulation results to show that the new algorithm has almost same ability to find feasible schedules. The third contribution is that we generalize this new algorithm by attempting to keep at least k processors busy if possible. We analyze the schedule length bound for the generalized algorithm. For tasks with equal computation times, we show that the bound is

$$min\left\{\frac{m}{k}+\sum_{j=2}^{k}\frac{1}{j},\quad r+\frac{17}{10}-\frac{(w+1)\cdot(r+\frac{7}{10}-\sum_{i=1}^{w}\frac{1}{i})}{m}-\frac{w}{m}\right\}$$

where $2 \leq k \leq m$ and w is the largest possible value such that $w < k$ and

$$\sum_{i=1}^{w}\frac{L_O}{i} \leq (r+\frac{7}{10})\cdot L_O+\frac{5}{2}.$$

For tasks with arbitrary computation times, we show the bound is

$$min\left\{\frac{m+1}{2}, r+1+\frac{m-2r-1}{k}\right\},$$

where $2 \leq k \leq m$.

As a by-product of our work, the result of this generalized algorithm also holds for the generalized list scheduling algorithm — a list scheduling algorithm that attempts to keep k processors busy if possible.

The remainder of the paper is organized as follows. Section 2 describes our problem, task model, terminology, assumptions, and some basic goals. Section 3 gives a detailed description of a particular heuristic scheduling algorithm. We analyze this heuristic scheduling algorithm in Section 4. Section 5 describes and analyzes the improved heuristic scheduling algorithm. In Section 6, the generalization of the improvement is presented and analyzed for tasks with arbitrary computation times as well as for tasks with the same computation times. We summarize the paper in Section 7.

2 The Models and Goals

In this section we define our problem, task model, terminology, and assumptions. These serve as the basis for the remaining sections. We explain why our problem is so hard, and the way we solve it. Finally we describe several goals of this works.

The scheduling problem we consider is characterized by a processor-resource-task model and a performance model.

The processor-resource-task model is described by the set $\mathcal{P}$ of processors, the set $\mathcal{R}$ of resources , and the set $\mathcal{T}$ of hard real-time tasks.

$\mathcal{P} = \{P_1, P_2, ..., P_m\}$ is a set of m identical processors in a homogeneous multiprocessor system. Each processor is capable of executing any task.

$\mathcal{R} = \{R_1, R_2, ..., R_r\}$ is a set of r types of resources such as data structures and buffers. They are *discrete and renewable.* A type of resource is discrete if it consists of a finite number of instances, e.g. one type of resource consists of 5 buffers with the same size. A type of resource is renewable if its total number of instances is always fixed and the resources are not consumed by the tasks. Each R_j may consist of multiple instances, and each instance can be used by tasks in two different modes: when in *shared mode*, several tasks can use the resource simultaneously; when in *exclusive mode*, only one task can use it at a time.

$\mathcal{T} = \{T_1, T_2, ..., T_n\}$ is a set of n tasks. Task T_i is characterized by the following:

- c_i — its worst case computation time,
- d_i — its deadline,
- $\vec{r_i} = (r_1, r_2, ..., r_r)$ — its resource requirement vector. Each r_j has two components: resource usage information and the number of instances requested for R_j. The resource usage information has one of three values: *not used, shared use*, and *exclusive use.* Besides these resource requirements, a task requires a processor.

We assume tasks are aperiodic, independent, nonpreemptable, and ready to start execution at the time of invocation of the scheduling algorithm. We also assume that the resources requested by a task are used throughout the task's execution time.

The performance model consists of two metrics: one describes the ability to find a feasible schedule and the other describes the quality of the final (feasible) schedule. In this paper, we use simulation to measure the ability to find a feasible schedule. The metric used is *mean success*

ratio (SR) defined in the following way. Given N task sets which are generated according certain distributions where each task set is known to have at least one feasible schedule, let N_A be the total number of task sets for which algorithm A finds a feasible schedule. Then

$$SR = \frac{N_A}{N}.$$

For the second metric, we use the schedule length bound as defined and explained earlier.

The above scheduling problem is a hard problem as reasoned below. Let us focus on one of the performance metrics and consider a subproblem. If there are three processors, one type of resource, and a set of tasks with the same computation time and deadline, it is NP-complete to find a feasible schedule [5] and it is NP-hard to find a minimum length feasible schedule [1]. This justifies the use of approximate (or heuristic) scheduling algorithms. As mentioned earlier, we choose the heuristic scheduling algorithm [13, 12] developed in the SPRING project as our basis.

There are several goals of this paper. Since the heuristic algorithm mentioned above has very good mean performance with respect to its ability to find feasible schedules, we want to know the quality of the feasible schedules generated by it, that is, we desire to determine its schedule length bound. We also want to search for a new algorithm with high SR, good schedule length bound, and good time complexity. Finally, we want to look at other possibilities to reduce the bound further on perhaps a restricted set of task parameters.

3 The Heuristic Scheduling Algorithm

In this section we compare scheduling to searching and present a heuristic function used to direct the search for a feasible schedule. The related data structures used in the heuristic algorithm are also described. At the end of the section, we present its pseudo code.

Scheduling a set of tasks to find a feasible schedule is actually a search problem. The structure of the search space is a search tree. An intermediate vertex of the search tree is a partial schedule, and a leaf, a terminal vertex, is a full schedule. It should be obvious that not all leaves, each a complete schedule, correspond to feasible schedules. If a feasible schedule is to be found, in the worst case, it might require an exhaustive search which is computationally intractable. Since in many real applications, a feasible schedule is time consuming to find and we need to find a feasible schedule quickly, we take a heuristic approach.

The heuristic scheduling algorithm tries to determine a full feasible schedule for a set of tasks in the following way. It starts at the root of the search tree which is an empty schedule and tries to extend the schedule (with a single task at a time) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. To this end, we use a heuristic function, H, which synthesizes various characteristics of tasks affecting real-time scheduling decisions to actively direct the scheduling to a plausible path. The heuristic function, H, is applied to each of the tasks that remain to be scheduled at each level of search. The task with the smallest value of function H is selected to extend the current schedule.

While extending the partial schedule at each level of search, the algorithm determines if the current partial schedule is *strongly-feasible* or not. A partial feasible schedule is said to be *strongly-feasible* if *all* the schedules obtained by extending this current schedule with any one of the remaining tasks are also feasible. Thus, if a partial feasible schedule is found not to be *strongly-feasible* because, say, task T_i misses its deadline when the current schedule is extended by T_i, then it is appropriate to stop the search since none of the future extensions involving task T_i will meet its deadline. In this case, a set of tasks can not be scheduled given the current partial schedule. (In the terminology of branch-and-bound techniques, the search path represented by the current partial schedule is *bound* since it will not lead to a feasible complete schedule.)

However, it is possible to backtrack to continue the search even after a non-strongly-feasible schedule is found. Backtracking is done by discarding the current partial schedule, returning to the previous partial schedule, and extending it by a different task. The task chosen is the one with the *second* smallest H value. Even though we allow backtracking, the overheads of backtracking are restricted either by restricting the maximum number of possible backtracks or by restricting the total number of evaluations of the H function.

In summary, given a particular H function, the algorithm works as follows: The algorithm starts with an empty partial schedule. Each step of the algorithm involves (1) determining that the current partial schedule is strongly-feasible, and if so (2) extending the current partial schedule by one task. This task is chosen by first applying the H function to all the tasks that are not in the current partial schedule and then determining the one with the least H value. This algorithm has a total of n steps, where the complexity of each step is given by the complexity of determining strong-feasibility and the complexity of H function evaluations. Both of these are linearly proportional to the number of tasks that remain to be scheduled. Each task may request up to r types of resources and one processor. Hence the overall complexity of this algorithm is

$(n + (n-1) + ... + 2) \cdot (r+1) = \mathrm{O}(n^2 r)$.

It is crucial that the above algorithm uses a good H function. The following H function combines task's deadline (d_i), task's computation time, and task's resource requirement information and has been shown to have the best performance among many other choices [12, 9, 10]

$$H(T_i) = d_i + W \cdot b_i$$

where W is an adjustable weight and b_i is the earliest time at which task T_i can begin execution given the current partial schedule, (i.e. at b_i all the resources needed by T_i are available). Calculation of b_i can be done along the lines of [10]. For the sake of brevity we omit the details here.

It is easy to observe that given a task's earliest start time, its finish time can be determined and thus the scheduling algorithm can decide if a task will finish by its deadline.

The above heuristic algorithm is called H_1 algorithm in the remainder of this paper. Here is its pseudo code.

THE H_1 SCHEDULING ALGORITHM

1: Compute $H(T_i) = d_i + W \cdot b_i$ for each unscheduled task T_i.
2: Select a task with the minimum H value to extend the current partial schedule.
3: If the current partial schedule is not strongly feasible, either backtrack or abort.
4: If there are tasks unscheduled, go to 1.

4 The Schedule Length Bound of the H_1 Algorithm

We present our first theorem in this section which based on three lemmas. Our analysis shows that the schedule length bound of the H_1 algorithm is m, and it is tight if $r \neq 0$. We first look at an example to provide an intuitive feeling for the bound and to demonstrate how deadlines can be chosen to satisfy both H_1 and the optimal schedule. This latter technique is used throughout later proofs.

Example 1: There are five tasks and one resource type. There are three processors and three instances of the resource type. The computation time and resource requirements are

shown by the following table.

task	T_1	T_2	T_3	T_4	T_5
computation time	1	1+ ϵ	1+ 2ϵ	ϵ	ϵ
#instances of the resource type	1	1	1	3	3

If the tasks are scheduled in the following order

$$(T_1, T_2, T_3, T_4, T_5),$$

then T_1, T_2, and T_3 run in parallel, and this schedule achieves the minimum schedule length which is

$$L_O = 1 + 4\epsilon.$$

If the tasks are scheduled by H_1 in the order of

$$(T_1, T_4, T_2, T_5, T_3),$$

then T_1, T_2, and T_3 run in sequential order, and the schedule length is

$$L_{H_1} = 3 + 5\epsilon.$$

As ϵ goes to zero the ratio of L_{H_1} over L_O asymptotically goes to 3, which is the number of processors.

All that is necessary now is to show that there exists a set of deadlines for tasks T_1 through T_5 such that H_1 would generate $(T_1, T_4, T_2, T_5, T_3)$, and in both the H_1 and optimal schedule all tasks make their deadlines. It can be checked that the following deadline setting satisfies these requirements.

task	T_1	T_2	T_3	T_4	T_5
deadline	1	$3 + W$	$4 + W$	2	$4 + W$

where W is the weight in the heuristic function given in section 3 and $W > 0$.

Let us now proceed with the formal properties of the H_1 heuristic. *List scheduling* is a well known greedy algorithm which will be discussed in detail in the next section and H_1 is a greedy algorithm too, although it is not as greedy as list scheduling. The property of greediness of H_1 is shown in lemma 1.

Lemma 1: For any schedule generated by the H_1 algorithm, let L_{H_1} be its schedule length. Then at least one processor is busy during the time interval $[0, L_{H_1}]$.

Proof If it is false, there exists a non-zero time interval $[t_1, t_2]$ in $[0, L_{H_1}]$ such that all processors are idle. Assume task T_j is scheduled by H_1 to start at time t_2. It means that b_j is equal to t_2 when T_j is picked by H_1. But at the same time, b_j is set to a time before or at t_1 because all resources are free at t_1. This is a contradiction which concludes that there is no such time interval in $[0, L_{H_1}]$ with all processors being idle. □

The method used in this paper to prove a tight schedule length bound for a given algorithm has two phases: (1) to show that there exists a bound and (2) to show that the bound can be reached. By combining the results of these two phases we can conclude that the algorithm has a tight bound. We first show that there exists a bound for H_1.

Lemma 2:

$$\frac{L_{H_1}}{L_O} \leq m.$$

Proof For any given processor-resource-task parameter setting, we have the following:

$$L_{H_1} \leq \sum_{i=1}^{n} c_i \leq m \cdot L_O.$$

This is because according to the lemma 1 at least one processor is busy until time is equal to L_{H_1}, and at most m processors are busy in an optimal schedule. □

Then we show that the bound is tight by presenting a particular worst case example.

Lemma 3: If $r \neq 0$, there exists a case x, which is a particular processor-resource-task parameter setting, such that,

$$\frac{L_{H_1}(x)}{L_O(x)} = m.$$

Proof If we find such an x, the lemma is proven.

Let $\mathcal{T}_1 = \{T_1, T_2, ..., T_m, T_1', T_2', ..., T_m'\}$, and $\mathcal{R}_1 = \{R\}$. Assume that there are m instances of the resource R. Let C be a constant and ϵ be a very small value. For each T_i, $c_i = C + i \cdot \epsilon$, and it requests one instance of the resource R for exclusive use, where $1 \leq i \leq m$. For each T_i', $c_i = \epsilon$, and it requests all m resource instances for exclusive use, where $1 \leq i \leq m$. Suppose the H_1 algorithm selects tasks from $\mathcal{T}_1$ in the following order:

$$(T_1, T_1', T_2, T_2', ..., T_m, T_m').$$

In fact, we can control the values of the tasks' deadlines, and hence the values of the function $H()$, such that H_1 generates the task execution order assumed.

The optimal algorithm selects tasks from $\mathcal{T}_1$ in the following order:

$$(T_1, T_2, ..., T_m, T_1', T_2', ..., T_m').$$

The bound is reached asymptotically, because in the optimal schedule, all m tasks T_i, $1 \leq i \leq m$, are scheduled to run in parallel, while in the schedule generated by the H_1 algorithm, these tasks are scheduled to run sequentially as they are blocked by the T_i' tasks one after another.

The schedule length for the H_1 algorithm is:

$$L_{H_1} = (C + 1 \cdot \epsilon + \epsilon) + (C + 2 \cdot \epsilon + \epsilon) + \cdots + (C + m \cdot \epsilon + \epsilon) = m \cdot C + \sum_{i=1}^{m}(i \cdot \epsilon) + m \cdot \epsilon.$$

The optimal schedule length is:

$$L_O = (C + m \cdot \epsilon) + m \cdot \epsilon.$$

So we have:

$$\begin{aligned} \frac{L_{H_1}}{L_O} &= \frac{m \cdot C + \sum_{i=1}^{m}(i \cdot \epsilon) + m \cdot \epsilon}{(C + m \cdot \epsilon) + m \cdot \epsilon} \\ &= \frac{m \cdot C + \frac{(1+m) \cdot m}{2} \cdot \epsilon + m \cdot \epsilon}{C + m \cdot \epsilon + m \cdot \epsilon} \\ &\rightarrow m, \ as \ \epsilon \rightarrow 0. \end{aligned}$$

□

By combining lemmas 2 and 3, the following theorem is derived.

Theorem 1: If $r \neq 0$, H_1 has a tight schedule length bound:

$$\frac{L_{H_1}}{L_O} \leq m.$$

5 Improving the Schedule Length Bound of the H_1 Algorithm

In this section we present an algorithm to improve schedule length bound of the H_1 algorithm. We demonstrate that it has good performance with respect to both criteria, namely, the average

success ratio, which measures the ability of finding feasible schedules, and the worst case schedule length bound, which measures one aspect of the quality of the feasible schedules. To motivate this algorithm, let reexamine list scheduling.

List scheduling is a very general greedy scheduling algorithm. It uses a *priority list* of tasks during scheduling. The priority list is created either beforehand or on-line, that is, it is either the static list constructed before the search starts or the dynamic list created at each search level in a search tree. Here we use the search paradigm again to help understand the scheduling process, as we did in Section 3. If there are no deadline constraints on tasks, every path of the search tree leads to a feasible schedule, though it may imply a different schedule length. With deadline constraints on tasks, it is not necessarily true. In both cases, list scheduling selects a task by scanning the priority list at each search level. The selected task is the highest priority task with the earliest start time among unscheduled tasks on the priority list. So the task's earliest start time is a primary criterion and the task's priority is a secondary criterion for resolving conflicts if there are several tasks having the same earliest start time. There are many list scheduling based algorithms, each uses a different method to generate a priority list. From the literature we present the following theorem on the schedule length bound for list scheduling. The theorem is rather strong in the sense that it applies to any list scheduling based algorithm.

Theorem 2: (M. R. Garey and R. L. Graham [3])
If $m \leq 2 \cdot r + 1$,

$$\frac{L_{List}}{L_O} \leq \frac{m+1}{2}.$$

If $m > 2 \cdot r + 1$,

$$\frac{L_{List}}{L_O} \leq r + 2 - \frac{2r+1}{m}.$$

Both bounds are tight.

The good schedule length bound of list scheduling is very attractive. Here we describe a list scheduling based algorithm named the *deadline-driven list scheduling* algorithm which uses the earliest deadline first (EDF) policy to create a static priority list.

DEADLINE-DRIVEN LIST SCHEDULING ALGORITHM

1: Create a priority list with EDF policy.

2: Select a task T_i with the earliest start time. Resolve the conflicts by the priority list if there are several tasks starting at the same time. Add the task to the current partial schedule.

3: If the current partial schedule is not strongly feasible, either backtrack or abort.

4: If there are tasks left unscheduled, go to 2.

This algorithm has a tight schedule length bound given by theorem 2, which is better than the bound of H_1. Unfortunately, it has a low success ratio, SR, as shown by simulation at the end of this section.

It is highly desirable to find an algorithm which has the same bound as list scheduling and has an average performance comparable to H_1's. So let us look at list scheduling and H_1 more carefully and identify their weak points. The weak point of the deadline-driven list scheduling algorithm is that it strictly follows a greedy resource usage rule, which will easily cause some tasks to miss their deadlines because the deadline is the secondary criterion for picking a next task. On the other hand, the weak point of the H_1 algorithm is that it strictly follows the minimum H value rule, which causes only one processor to be busy in some situations. A new heuristic scheduling algorithm named H_2 algorithm, described next, avoids these weak points by keeping two processors busy whenever possible.

The H_2 algorithm maintains a variable called t_c which divides the current partial schedule into two portions: one portion is before t_c and the other is after t_c. t_c is set to the maximum possible value satisfying the following: In any sub-interval $[x, y]$ of $[0, t_c)$, (1) at least two processors are busy, or (2) only one processor is busy but the resource requirements of the task scheduled on the processor are such that no other task (scheduled to start after y or yet unscheduled) can be scheduled to start before y. Note that it may be possible to schedule another task, which is not in the current partial schedule, prior to t_c if the resources needed for it are available before t_c.

THE H_2 SCHEDULING ALGORITHM

1: Compute $H(T_i) = d_i + W \cdot b_i$ for each unscheduled task T_i.

2: Determine t_c for the current partial schedule. Then using t_c to partition the unscheduled tasks into two sets:

$$S_1 = \{T_i : b_i \leq t_c\}, \quad S_2 = \{T_i : b_i > t_c\}.$$

Select a task with the minimum H value in S_1. (S_1 is not empty by the definition of t_c.) Add the task to the current partial schedule.

3: If the current partial schedule is not strongly feasible, either backtrack or abort.

4: If there are tasks unscheduled, go to 1.

Both H_1 and H_2 have the same time complexity, because step 2 of H_2 can be done in $O(n)$ as in H_1. To analyze the schedule length bound of H_2, we establish lemmas 4 to 6 first. The result of lemma 4 is used for proving lemma 5. Lemma 5 shows that there exists a schedule length bound for H_2 and by finding a particular worst case example, lemma 6 shows that the bound can be reached asymptotically.

For any feasible schedule generated by the H_2 algorithm, we define:

$$\mathcal{B}_i = \{\text{all time intervals with exactly } i \text{ processors busy }\}, \text{ and}$$
$$I_i = \text{the sum of the length of all time intervals in } \mathcal{B}_i,$$

where $1 \le i \le m$.

Example 2: Figure 1 shows a schedule of ten tasks on three processors. The resources used by the tasks are omitted in the figure. The schedule is partitioned into intervals defined above.

$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_3$	I_1	I_2	I_3
{ [3,5], [12,13] }	{ [0,3], [11,12], [13,20] }	{ [5,11] }	3	11	6

Notice that lemma 1 is also true for the H_2 algorithm, which generates the schedule with at least one processor being busy in any time interval in $[0, L_{H_2}]$.

Lemma 4:

$$I_1 \le L_O.$$

Proof Let $SCHED$ be any schedule generated by the H_2 algorithm. Notice that a task can not be involved in two or more different time intervals in $\mathcal{B}_1$ because of the greedy nature of H_2. Now assume that $I_1 > L_O$. Then there exist at least two time intervals in $\mathcal{B}_1$ with their tasks being scheduled in parallel in any optimal schedule. So the tasks in these two time intervals have no conflicts on their resource requirements, and therefore H_2 (because it is greedy) must schedule them in parallel. It means that $SCHED$ is not the schedule generated by the H_2 algorithm. This is a contradiction which concludes that $I_1 \le L_O$. □

We now show that there exists a schedule length bound for H_2.

Lemma 5: For any m and r,

$$\frac{L_{H_2}}{L_O} \leq \frac{m+1}{2}.$$

Proof Because I_1 accounts for one processor being busy and the remaining I_i account for at least two processors being busy,

$$I_1 + 2 \cdot \sum_{i=2}^{m} I_i \leq \sum_{i=1}^{n} c_i.$$

The schedule length of H_2 is L_{H_2}. So $\sum_{i=2}^{m} I_i = L_{H_2} - I_1$.
Since, $\sum_{i=1}^{n} c_i \leq m \cdot L_O$,

$$I_1 + 2 \cdot (L_{H_2} - I_1) \leq \sum_{i=1}^{n} c_i \leq m \cdot L_O.$$

Rewriting the above inequality:

$$\begin{aligned} 2 \cdot L_{H_2} - I_1 &\leq m \cdot L_O \\ 2 \cdot L_{H_2} - L_O &\leq m \cdot L_O, \qquad \text{by lemma 4} \\ 2 \cdot L_{H_2} &\leq (m+1) \cdot L_O \\ \frac{L_{H_2}}{L_O} &\leq \frac{m+1}{2}. \end{aligned}$$

□

We now show that the bound is tight by presenting a particular example.

Lemma 6: For any m and $r \neq 0$, there exists a case x, which is a particular processor-resource-task parameter setting, such that,

$$\frac{L_{H_2}(x)}{L_O(x)} = \frac{m+1}{2}.$$

Proof If we find such an x, the lemma is proven.

Let $\mathcal{T}_2 = \{T_1, T_2, \cdots, T_m, T_1', T_2', \cdots, T_m', T_1'', T_2'', \cdots, T_m''\}$, and $\mathcal{R}_2 = \{R\}$. Assume that there are $2m$ instances of R. Let C be a constant and ϵ be a very small value. Each T_i and T_i' has $c_i = c_i' = C + i \cdot \epsilon$, $1 \leq i \leq m$. All T_i and T_i' request one instance of R for exclusive

use except for T_m and T_m' which request $m+1$ instances of R for exclusive use. For each T_i'', $c_i'' = \epsilon$, and it requests all $2m$ resource instances of R for exclusive use, where $1 \leq i \leq m$.

We force H_2 to select tasks in $\mathcal{T}_2$ in the following order by adjusting their deadlines:

$$(T_1, T_1', T_1'', T_2, T_2', T_2'', \cdots, T_m, T_m', T_m'').$$

The optimal algorithm selects $\mathcal{T}_2$ in the following order:

$$(T_1, T_2, \cdots, T_m, T_1', T_2', \cdots, T_m', T_1'', T_2'', \cdots, T_m'').$$

The schedule length for H_2 is:

$$\begin{aligned}
L_{H_2} &= (C + 1 \cdot \epsilon + \epsilon) + (C + 2 \cdot \epsilon + \epsilon) + \cdots + (C + (m-1) \cdot \epsilon + \epsilon) + (2(C + m \cdot \epsilon) + \epsilon) \\
&= (m+1) \cdot C + 2 \cdot m \cdot \epsilon + \sum_{i=1}^{m} (i \cdot \epsilon) \\
&= (m+1) \cdot C + 2 \cdot m \cdot \epsilon + \frac{m \cdot (m+1)}{2} \cdot \epsilon.
\end{aligned}$$

The optimal schedule length is:

$$L_O = 2 \cdot (C + m \cdot \epsilon) + m \cdot \epsilon.$$

So we have:

$$\frac{L_{H_2}}{L_O} = \frac{(m+1) \cdot C + 2 \cdot m \cdot \epsilon + \frac{m \cdot (m+1)}{2} \cdot \epsilon}{2 \cdot (C + m \cdot \epsilon) + m \cdot \epsilon} \rightarrow \frac{m+1}{2}, \ as\ \epsilon \rightarrow 0.$$

□

By combining lemmas 2 and 3, the following theorem is derived.

Theorem 3: If $r \neq 0$, H_2 has a tight schedule length bound:

$$\frac{L_{H_2}}{L_O} \leq \frac{m+1}{2}.$$

In the remainder of this section, we use simulation to evaluate performance given by the success ratios of three algorithms: H_1, H_2 and the deadline-driven list scheduling algorithm (see Figures 2 and 3). Clearly, what we are striving for is a scheduling algorithm that is able

to find a feasible schedule for a set of tasks, if such a schedule exists. Obviously, a heuristic algorithm cannot be guaranteed to achieve this. However, one heuristic algorithm can be considered better than another, if given a number of task sets for which feasible schedule exist, the former is able to find feasible schedules for more task sets than the latter. This is the basis for our simulation study. Ideally, we would like to come up with a number of task sets, each of which is known to have a feasible schedule. Unfortunately, given an arbitrary task set, only an exhaustive search can reveal whether the tasks in this task set can be feasibly scheduled.

Therefore we take a different approach in our study here. We use the task generator which is used in [9, 10], which can generate schedulable task sets where the number of tasks in a task set can be very large. Also the tasks are generated to guarantee the (almost) total utilization of the processors. The schedule generated by the task generator is used only for the purpose of generating a feasible set of tasks which is then input to the scheduling algorithm, i. e., the scheduling algorithms have no knowledge of the schedule itself but are only given the tasks and their requirements. The following are the parameters used to generate the task set:

-Probability that a task uses a resource, Use_P.

-Probability that a task uses a resource in shared mode, Share_P.

-The minimum computation time of tasks, Min_C.

-The maximum computation time of tasks, Max_C.

-The schedule length, L, which controls the length of a simulation.

The schedule generated by this task set generator is in the form of a matrix which has $m + r$ columns and L rows. The first m columns represent m processors. The remaining columns represent r resource types with one instance for each resource type. Each row represents a time unit. The task set generator starts with an empty matrix, then generates a task by selecting one of the m processors with the earliest available time and then requests the r resources according to the probabilities specified in the generation parameters. The generated task's processing time is randomly chosen using a uniform distribution between the minimum processing time and the maximum processing time. The task set generator marks on the matrix that the processor and resources required by the task are used up for a number of time units equal to the task's computation time starting from the aforementioned earliest available time of the processor. The task set generator generates tasks until the remaining unused time units of each processor, up to L, is smaller than the minimum processing time

of a task, which means that no more tasks can be generated to use the processors. Then the largest finish time of a generated task becomes the task set's *shortest completion time*, SC. As a result, we generate tasks according to a very tight schedule without leaving any usable time units on the m processors between 0 and SC. However, there may be some empty time units in the r resources. So the generated matrix can be treated as the schedule generated by an optimal scheduling algorithm with respect to the feasibility and the schedule length.

So far we have discussed how task resource requirements and computation times are determined. The issue of choosing task deadlines without any bias is addressed now. In order to exercise the scheduling algorithms in scenarios that have different levels of scheduling difficulty, we choose the deadline of a task in the task set randomly between $(1+R)\cdot f_i$ and $(1+R)\cdot SC$, where f_i is the task's completion time in the above matrix and R (the Relaxation Factor) is a simulation parameter indicating the tightness of the deadlines. As we increase the value of R, it is not difficult to see that the scheduler has a better and better chance to guarantee a task set. Because of this unbiased generation of task sets we believe that the resulting task sets can be used to evaluate the heuristic algorithms rigorously.

The simulation results are obtained by running 200 task sets. The number of processors is 5 and the number of other types of resources is 12. There is one instance for each type of resource. A task's computation time is randomly chosen between Min_C (10) and Max_C (40). There are two parameters to control task's resource requests. One is resource use probability, Use_P, which may vary from 0.1 to 0.7. The other is shared probability, Shared_P, set to 0.5. There are about 40 tasks in each task set.

Figure 2 shows that the effect of R on the success ratio. The performance of H_1 and H_2 is much better than the performance of list scheduling, e.g. at $R = 0.2$ and Use_$P = 0.7$, SR increases from 57% for list scheduling to 81% and 79% for the H_1 and H_2 algorithm, respectively, and at $R = 0.2$ and Use_$P = 0.3$, SR increases from 20% for list scheduling to 62% and 61% for the H_1 and H_2 algorithm, respectively. Figure 3 shows the effect of resource contention by changing the probability of task's resource request. Again, the performance of H_1 and H_2 are close and far better than the performance of list scheduling, e.g. when Use_$P = 0.5$, SR increases from 28% for list scheduling to 62% and 60% for the H_1 and H_2 algorithm, respectively. These show that whereas H_2 algorithm has close performance to H_1, it is much better than the deadline-driven list scheduling algorithm.

6 A Generalized Heuristic Scheduling Algorithm and Its Schedule Length Bound

As mentioned in the last section, the H_2 algorithm basically keeps at least two processors busy whenever possible, thus the schedule length bound is reduced from m to $(m+1)/2$. In this section, we generalize this idea further to keep at least k processors busy whenever possible, but otherwise keep as many processors busy as possible.

Some interesting aspects about the generalization are summarized here: (1) the generalized algorithm slightly reduces the worst case schedule length bound for task sets with non-uniform tasks, if $m > 2r+1$; (2) it reduces the schedule length bound for task sets with uniform tasks. (We define *uniform tasks* as tasks with the same computation time.)

6.1 The H_k Algorithm for Non-Uniform Tasks

The H_k algorithm maintains a variable called t_{c_k} which divides a partial schedule into two portions: one portion is before t_{c_k} and the other is after t_{c_k}. t_{c_k} is set to the maximum possible value that satisfies the following: In any sub-interval $[x, y]$ of $[0,\ t_{c_k})$, (1) at least k processors are busy, or (2) less than k processors are busy, but the resource requirements of the tasks scheduled in $[x, y]$ are such that no other task (scheduled to start after y or yet unscheduled) can be scheduled to start before y. Note that it may be possible to schedule another task, which is not in the current partial schedule, prior to t_{c_k} if the resources needed for it are available before t_{c_k}.

THE H_k SCHEDULING ALGORITHM

1:Compute $H(T_i) = d_i + W \cdot b_i$ for each unscheduled task T_i.

2:Determine t_{c_k} for the current partial schedule. Partition unscheduled tasks into three sets:

$$S_1 = \{T_i : b_i + c_i \leq t_{c_k}\}, \quad S_2 = \{T_i : (b_i \leq t_{c_k}) \& (b_i + c_i > t_{c_k})\}, \quad S_3 = \{T_i : b_i > t_{c_k}\}.$$

S_1 is the set of unscheduled tasks which can complete before t_{c_k}.

S_2 is the set of unscheduled tasks which can start before t_{c_k} but complete after t_{c_k}.

S_3 is the set of unscheduled tasks which can start only after t_{c_k}.

Sort S_1, S_2, and S_3 by heuristic values in non-decreasing order.

Select a task with the minimum H value in S_1 if it is not empty.

Otherwise go through S_2 (which is sorted) selecting a subset of tasks with minimal possible H value such that this subset and the tasks in the partial schedule cause up to k tasks to be scheduled to run in parallel at time t_{c_k}. (S_1 and S_2 are not empty at the same time by the definition of t_{c_k}.) Add the task/tasks to the current partial schedule.

3:If the current partial schedule is not strongly feasible, either backtrack or abort the algorithm.

4:If there are tasks left unscheduled, go to 1.

The complexity of H_k is $O(n^{k+1}r^k)$, where $2 \leq k \leq m$, because in step 2, if all the processors are idle at the time t_{c_k}, the algorithm may need to worry about finding up to k tasks which are scheduled to execute at t_{c_k}. Each task can request up to r resource types. Therefore, the time complexity for step 2 is $O(n^k r^k)$. The H_k algorithm may loop n times. This leads to an $O(n^{k+1}r^k)$ complexity.

Schedule length bound of H_k is analyzed in the following way. Lemmas 7 and 8 derive the bound for $m \leq 2r+1$. Lemmas 9, 10, 11, and 12 derive the bound for $m > 2r+1$. Combining the bounds for these two cases, we derive theorem 4.

Lemma 7: For $m \leq 2r+1$ and $2 \leq k \leq m$,

$$\frac{L_{H_k}}{L_O} \leq \frac{m+1}{2}.$$

Proof The inequality follows immediately from lemma 5. □

Lemma 8: For $m \leq 2r+1$ and $2 \leq k \leq m$, there exists a case x, which is a particular processor-resource-task parameter setting, such that,

$$\frac{L_{H_k}(x)}{L_O(x)} = \frac{m+1}{2}.$$

Proof If we find such an x, the lemma is proven.

We consider two cases.

Case 1: $2 \leq m \leq r+1$.

Let

$$\mathcal{T}_3 = \{T_i, T_i^{'} : 1 \leq i \leq m-1\} \cup \{T^{''}\} \cup \{T_i^j : 1 \leq i \leq m-1,\ 1 \leq j \leq k-2\}$$

and

$$\mathcal{R}_3 = \{R_1, R_2, ..., R_r\}.$$

There are two instances of each type of resource. Further let C be a constant. Each T_i and $T_i^{'}$ has the same computation time C and requests one instance of R_i for exclusive use and one instance of all other resource types except for R_i for shared use, where $1 \leq i \leq m-1$. $T^{''}$ needs computation time of $2C$ and one instance of every resource type for shared use. Each T_i^j has $c_i^j = \epsilon$, and requests one instance of every resource type except R_i for shared use, where $1 \leq i \leq m-1$, $1 \leq j \leq k-2$. These small tasks are used for running in parallel with two large tasks in the schedule generated by the H_k algorithm to keep k processors busy.

We force H_k to select the tasks in the following order by choosing their deadlines properly:

$$(T_1, T_1^{'}, T_1^1, T_1^2, ..., T_1^{k-2}, T_2, T_2^{'}, T_2^1, T_2^2, ..., T_2^{k-2}, ..., T_{m-1}, T_{m-1}^{'}, T_{m-1}^1, T_{m-1}^2, ..., T_{m-1}^{k-2}, T^{''}).$$

The optimal algorithm may select tasks from $\mathcal{T}_3$ in the following order:

$$(T^{''}, T_1, T_2, ..., T_{m-1}, T_1^{'}, T_2^{'}, ..., T_{m-1}^{'}, T_1^1, T_1^2, ..., T_1^{k-2}, T_2^1, T_2^2, ..., T_2^{k-2}, ..., T_m^1, T_m^2, ..., T_m^{k-2}).$$

In the optimal schedule, all T_i tasks are scheduled to run in parallel and all $T_i^{'}$ tasks are also scheduled to execute in parallel, they are scheduled to run in parallel with $T^{''}$, and all remaining tasks can be scheduled within a time interval $(m-1) \cdot \epsilon$. Then the schedule length for the optimal schedule is $L_O = 2 \cdot C + (m-1) \cdot \epsilon$. In the schedule generated by the H_k algorithm, the two large tasks, T_i and $T_i^{'}$, are scheduled to execute in parallel with $(k-2)$ small tasks, T_i^j, where where $1 \leq j \leq k-2$ and $1 \leq i \leq m-1$. Then $T^{''}$ follows because it is blocked by every pair of T_i and $T_i^{'}$. So its schedule length is $L_{H_2} = (m-1) \cdot C + 2 \cdot C$.

We have

$$\frac{L_{H_k}}{L_O} = \frac{(m-1) \cdot C + 2 \cdot C}{2 \cdot C + (m-1) \cdot \epsilon} \rightarrow \frac{m+1}{2}, \; as \; \epsilon \rightarrow 0.$$

Case 2: $r+1 \leq m \leq 2r+1$.

For simplicity, here we assume that each task can ask for a fraction of a resource. (This corresponds to the fraction of the number instances of a resource type required by a task.)

For an arbitrary integer s, let

$$\mathcal{T}_4 = \{T_0\} \cup \{T_{i,j}, T_{i,j}^{'} : 1 \leq i \leq m-1, 1 \leq j \leq s\}$$

$$\cup\{T_{i,j}^l : 1 \le i \le m-1, 1 \le j \le s, 1 \le l \le k-2\}$$

and

$$\mathcal{R}_4 = \{R_1, R_2, ..., R_r\}.$$

$c_0 = 2s$, $c_{i,j} = c_{i,j}^{'} = 1$, $c_{i,j}^l = \epsilon$, where $1 \le i \le m-1, 1 \le j \le s, 1 \le l \le k-2$.

T_0 requests $\epsilon/(m-1)^{2s-1}$ of every resource type for exclusive use.

For $1 \le i \le r$ and $1 \le j \le s$, $T_{i,j}$ requests $1 - \epsilon/(m-1)^{2s-2j} - \epsilon/(m-1)^{2s+sm-jr-i}$ of R_i for exclusive use and $\epsilon/(m-1)^{2s-2j}$ of all other resource types except for R_i for exclusive use.

For $r+1 \le i \le r$ and $1 \le j \le s$, $T_{i,j}$ requests $\epsilon/(m-1)^{2s-2j}$ of every resource type for exclusive use.

For $1 \le i \le m-r-1$ and $1 \le j \le s$, $T_{r+i,j}^{'}$ requests $1 - \epsilon/(m-1)^{2s-2j+1}$ of R_i for exclusive use and $\epsilon/(m-1)^{2s-2j}$ of all other resource types except for R_i for exclusive use.

For $1 \le i \le r$ and $1 \le j \le s$, $T_{i,j}^{'}$ requests $\epsilon/(m-1)^{2s-2j}$ of every resource type for exclusive use.

For $1 \le i \le m-1$, $1 \le j \le s$, and $1 \le l \le k-2$, $T_{i,j}^l$ requests $\epsilon/(m-1)^{2s+sm-jr-i}$ of every resource type shared use.

By the above task's parameter setting, we are able to find an optimal schedule such that T_0 is scheduled in parallel with $2s$ subgroups of $\{T_{i,j}\}$ and $\{T_{i,j}^{'}\}$, each subgroup contains exactly $m-1$ tasks, and all remaining small tasks are scheduled at the end of the schedule; while in the schedule generated by H_k, tasks are scheduled in subgroups of k tasks, with one from $\{T_{i,j}\}$, one from $\{T_{i,j}^{'}\}$, and $k-2$ from $\{T_{i,j}^l\}$, such that after a subgroup has been scheduled in a partial schedule, it blocks all remaining unscheduled tasks because of resource contention, and T_0 is scheduled in the last with several unscheduled $\{T_{i,j}\}$ and $\{T_{i,j}^{'}\}$ tasks as described in detail in the following.

We force H_k to select the tasks in the following order by adjusting their deadlines:

$$(X_1, X_2, \cdots, X_s, Y_1, Y_2, \cdots, Y_{s-1}, Z),$$

where $X_i = (U_{1,i}, U_{2,i}, \cdots, U_{r,i}), \ \ 1 \le i \le s$

$$U_{j,i} = (T_{j,i}, T_{j,i}^{'}, T_{j,i}^1, T_{j,i}^2, \cdots, T_{j,i}^{k-2}), \ \ 1 \le i \le s, 1 \le j \le r$$

$$Y_i = (V_{1,i}, V_{2,i}, \cdots, V_{m-r-1,i}), \quad 1 \le i \le s-1,$$

$$V_{j,i} = (T'_{r+j,i}, T_{r+j,i+1}, T^1_{r+j,i+1}, T^2_{r+j,i+1}, \cdots, T^{k-2}_{r+j,i+1}),$$
$$1 \le i \le s-1, \quad 1 \le j \le m-r-1,$$

$$Z = (T_0, T_{r+1,1}, T_{r+2,1}, \cdots, T_{m-1,1}, T'_{r+1,s}, T'_{r+2,s}, \cdots, T'_{m-1,s}, Z'),$$

$$Z' = (T^1_{r+1,1}, \cdots, T^{k-2}_{r+1,1}, T^1_{r+2,1}, \cdots, T^{k-2}_{r+2,1}, \cdots\cdots, T^1_{m-1,1}, \cdots, T^{k-2}_{m-1,1}).$$

The optimal algorithm may select tasks from $\mathcal{T}_4$ in the following order:

$$(T_0, W_1, W_2, \cdots, W_s, W'_1, W'_2, \cdots, W'_s, Q),$$

where

$$W_i = (T_{1,i}, T_{2,i}, \cdots, T_{m-1,i}), \quad 1 \le i \le s,$$

$$W'_i = (T'_{1,i}, T'_{2,i}, \cdots, T'_{m-1,i}), \quad 1 \le i \le s,$$

and Q includes all remaining small tasks in arbitrary order.

It can be checked that H_k schedules $k-2$ small tasks from $T^l_{i,j}$, and two other tasks, one from $T_{i,j}$ and one from $T'_{i,j}$, together such that k processors busy condition satisfied and the remaining unscheduled tasks are blocked. There are s groups of X_i, each X_i contains r subgroups of k tasks, and the schedule length of all X_i's is sr. There are $s-1$ groups of Y_i, each Y_i contains $m-r-1$ subgroups of k tasks, and the schedule length of all Y_i's is $s(m-r-1)$. The schedule length of Z is $2s$. So

$$L_{H_k} = sr + (s-1)(m-r-1) + 2s = s(m+1) - (m-r-1).$$

The length of the optimal schedule is

$$L_O = 2s + \epsilon \left\lceil \frac{(m-1)(s-1)(k-2)}{m} \right\rceil.$$

We then have

$$\frac{L_{H_k}}{L_O} = \frac{s(m+1) - (m-r-1)}{2s + \epsilon \left\lceil \frac{(m-1)(s-1)(k-2)}{m} \right\rceil} \to \frac{m-1}{2},$$

when s is very large. □

Now we turn our attention to the bound when $m > 2r+1$.

Lemma 9: (M. R. Garey and R. L. Graham [3])

If m is unbounded for an arbitrary r

$$\frac{L_{List}}{L_O} \leq r + 1.$$

A schedule generated by H_k can be partitioned into $\mathcal{B}_i$, $1 \leq i \leq m$, as defined earlier. According to the H_k algorithm, $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_{k-1}$ consists of intervals with the following property. Each such interval consumes a certain amount of resources, none of the tasks scheduled after this interval can be scheduled to start in this interval because of resource contentions. Intuitively, when the H_k algorithm schedules tasks in the interval sets $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_{k-1}$, it operates like list scheduling in that one cannot add one more task to run in parallel with tasks in these intervals. When H_k schedules tasks in the interval sets $\mathcal{B}_k$, $\mathcal{B}_{k+1}$, $\cdots$, and $\mathcal{B}_{m-1}$, it works like H_1 which strictly uses the rule of the minimum heuristic value first. So, if we compare the schedule generated by H_k and the schedule generated by list scheduling, we derive the following inequality:

$$\frac{\sum_{i=1}^{k-1} I_i}{L_O} \leq \frac{L_{List}}{L_O} \leq r + 1.$$

This leads to the following lemma.

Lemma 10: For any schedule generated by H_k and an arbitrary r,

$$\frac{\sum_{i=1}^{k-1} I_i}{L_O} \leq r + 1,$$

where $2 \leq k \leq m$.

Lemma 11: For $m > 2r + 1$ and $2 \leq k \leq m$,

$$\frac{L_{H_k}}{L_O} \leq r + 1 + \frac{m - 2r - 1}{k}.$$

Proof For any schedule generated by H_k, we have

$$\sum_{i=k}^{m} I_i = L_{H_k} - \sum_{i=1}^{k-1} I_i,$$

and

$$\sum_{i=1}^{m} I_i \leq \sum_{i=1}^{n} c_i.$$

In an optimal schedule,

$$\sum_{i=1}^{n} c_i \leq m \cdot L_O.$$

Combining the above two inequalities, we have

$$\sum_{i=1}^{m} I_i \leq m \cdot L_O.$$

Further,

$$I_1 + \sum_{i=2}^{k-1} i \cdot I_i + \sum_{i=k}^{m} i \cdot I_i \leq m \cdot L_O,$$

$$I_1 + 2\sum_{i=2}^{k-1} I_i + k \cdot \sum_{i=k}^{m} I_i \leq m \cdot L_O,$$

$$I_1 + 2\sum_{i=2}^{k-1} I_i + k \cdot (L_{H_k} - \sum_{i=1}^{k-1} I_i) \leq m \cdot L_O,$$

$$k \cdot L_{H_k} \leq m \cdot L_O + I_1 + (k-2) \cdot \sum_{i=1}^{k-1} I_i.$$

By lemma 4 and 10, we have

$$k \cdot L_{H_k} \leq m \cdot L_O + L_O + (k-2) \cdot (r+1).$$

Then

$$\frac{L_{H_k}}{L_O} \leq r + 1 + \frac{m - 2r - 1}{k}.$$

□

Lemma 12: For $m > 2r + 1$ and $2 \leq k \leq m$, there exists a case x, which is a particular processor-resource-task parameter setting, such that,

$$\frac{L_{H_k}}{L_O} \leq r + 1 + \frac{m - 2r - 1}{k}.$$

Proof If we find such an x, the lemma is proven.

Again we assume that each task can ask for a fraction of a resource. For an arbitrary integer s', let $s = s'k$ and

$$\mathcal{T}_5 = \{T_0\} \cup \{T_{i,j}, : 1 \leq i \leq m-1, 1 \leq j \leq s\} \cup \{T^l_{i,j} : 1 \leq i \leq r, 1 \leq j \leq s-1, 1 \leq l \leq k-2\}$$

and

$$\mathcal{R}_5 = \{R_1, R_2, ..., R_r\}.$$

Let $c_0 = s$, $c_{i,j} = 1$, where $1 \leq i \leq m-1, 1 \leq j \leq s, 1 \leq l \leq k-2$.

$c^l_{i,j} = \epsilon$ where $1 \leq i \leq r, 1 \leq j \leq s, 1 \leq l \leq k-2$.

T_0 requests $\epsilon/(m-1)^{s-1}$ of every resource type for exclusive use.

For $1 \leq i \leq r$ and $1 \leq j \leq s$, $T_{i,j}$ requests $1 - \epsilon/(m-1)^{s-j-1} - \epsilon/(m-1)^{s+(s-1)r-(s-1)j+i}$ of R_i for exclusive use and $\epsilon/(m-1)^{s-j}$ of all other resource types for exclusive use.

For $r+1 \leq i \leq m-1$ and $1 \leq j \leq s$, $T_{i,j}$ requests $\epsilon/(m-1)^{s-j}$ of every resource type for exclusive use.

For $1 \leq i \leq r$, $1 \leq j \leq s-1$, and $1 \leq l \leq k-2$, $T^l_{i,j}$ requests $\epsilon/(m-1)^{s+(s-1)r-(s-1)j+i}$ of every resource type for shared use.

By the above task's parameter setting, we are able to find an optimal schedule such that T_0 is scheduled in parallel with s subgroups of $\{T_{i,j}\}$ tasks, each subgroup contains exactly $m-1$ tasks, and then all remaining small ϵ tasks are scheduled later in the schedule; while in the schedule generated by H_k, tasks are scheduled in subgroup of k tasks, with subgroups of all k tasks come from $\{T_{i,j}\}$ in the beginning of the schedule, followed by subgroups which have two tasks from $\{T_{i,j}\}$ tasks and $k-2$ from $\{T^l_{i,j}\}$, then T_0 is scheduled at the end of the schedule with several unscheduled $\{T_{i,j}\}$ as described in detail in the following.

We force H_k to select the tasks in the following order by adjusting their deadlines:

$$(X_1, X_2, \cdots, X_{m-2r-1}, Y_1, Y_2, \cdots, Y_r, Z),$$

where $X_i = (U_{i,1}, U_{i,2}, \cdots, U_{i,s'}), \;\; 1 \leq i \leq m-2r-1$,
$U_{i,j} = (T_{2r+i,(j-1)k+1}, T_{2r+i,(j-1)k+2}, \cdots, T_{2r+i,(j-1)k+k}, \quad 1 \leq i \leq m-2r-1, \;\; 1 \leq j \leq s'$,

$$Y_i = (V_{i,1}, V_{i,2}, \cdots, V_{i,s-1}), \;\; 1 \leq i \leq r, \quad V_{i,j} = (T_{i,j}, T_{r+i,j+1}, T^1_{i,j}, T^2_{i,j}, \cdots, T^{k-2}_{i,j}),$$
$$1 \leq i \leq r, \;\; 1 \leq j \leq s-1,$$

$$Z = (T_0, T_{r+1,1}, T_{r+2,1}, \cdots, T_{2r,1}, T_{1,s}, T_{2,s}, \cdots, T_{r,s}).$$

The optimal algorithm may select tasks from $\mathcal{T}_5$ in the following order:

$$(T_0, W_1, W_2, \cdots, W_s, Q)$$

where

$$W_i = (T_{1,i}, T_{2,i}, \cdots, T_{m-1,i}), \quad 1 \le i \le s,$$

and Q includes all small ϵ tasks in arbitrary order.

The schedule length of H_k consists of three parts, all X_i contribute $(m-2r-1)s'$, all Y_j contribute $(s-1)r$, and Z contribute s:

$$L_{H_k} = (m-2r-1)s' + (s-1)r + s$$

and the length of the optimal schedule is

$$L_O = s + \epsilon \left\lceil \frac{(k-2)(s-1)r}{m} \right\rceil.$$

So,

$$\frac{L_{H_k}}{L_O} = \frac{(m-2r-1)s' + (s-1)r + s}{s + \epsilon \left\lceil \frac{(k-2)(s-1)r}{m} \right\rceil} \to r + 1 + \frac{m-2r-1}{k}, \quad s \to \infty.$$

□

Combining the above results, we derive the following theorem.

Theorem 4: For any r and m, H_k has a tight schedule length bound:

$$\frac{L_{H_k}}{L_O} \le min\left\{\frac{m+1}{2}, r + 1 + \frac{m-2r-1}{k}\right\},$$

where $2 \le k \le m$.

6.2 The H_k Algorithm for Uniform Tasks

For uniform tasks, the situation is much better. Let us look at an example first.

Example 3: Given $m = 4$, $k = 3$, and $r = 4$, with the following setting. There are three instances of R_1 and R_2, two instances of R_3, and one instances of R_4. There are 24 tasks:

$$\{T_{i,j} : 1 \le i \le 4, 1 \le j \le 6\}$$

All tasks have one unit computation time.

For $1 \leq j \leq 6$, each $T_{1,j}$ requests one instance of R_1 for exclusive use; each $T_{2,j}$ requests one instance of R_2 for exclusive use; each $T_{3,j}$ requests one instance of R_1, one instance of R_2, and one instance of R_3 for exclusive use; each $T_{4,j}$ requests one instance of R_1, one instance of R_2, one instance of R_3, and one instance of R_4.

We force the H_3 algorithm to schedule the tasks in the following order by adjusting their deadlines:

$$(T_{1,1}, T_{1,2}, \cdots, T_{1,6}; T_{2,1}, T_{2,2}, \cdots, T_{2,6}; \cdots; T_{4,1}, T_{4,2}, \cdots, T_{4,6}).$$

The optimal algorithm uses the following order:

$$(T_{1,1}, T_{2,1}, T_{3,1}, T_{4,1}; T_{1,2}, T_{2,2}, T_{3,2}, T_{4,2}; \cdots; T_{1,6}, T_{2,6}, T_{3,6}, T_{4,6}).$$

Then it is easy to verify that the length of the schedule generated by the H_3 algorithm is 13 time units, and the length of the schedule generated by an optimal scheduling algorithm is 6 time units as shown in Figure 4. Thus we have:

$$\frac{L_{H_3}(x)}{L_O(x)} = \frac{4}{3} + (\frac{1}{2} + \frac{1}{3}) = \frac{13}{6}.$$

In the above example, after all $T_{1,j}$ and $T_{2,j}$ are scheduled by H_3, there is no other subset of three tasks left in the remaining unscheduled tasks such that the subset can be scheduled in parallel. After all the $T_{3,j}$ are scheduled next, the remaining six tasks can only be scheduled in sequential order. As it turns out, this is the worst case schedule length as proved via lemma 15.

Let ϵ be a very small quantity and

$$\mathcal{S} = \{a_1, a_2, \cdots, a_k, a_{k+1}, \cdots, a_m\} = \{\frac{1}{k}, \frac{1}{k}, \cdots, \frac{1}{k}, \frac{1}{k-1}, \cdots, \frac{1}{3}, \frac{1}{2}, \frac{1}{2} + \epsilon\}.$$

Let us consider a minimum bin packing problem that takes $\mathcal{S}$ as the input and tries to pack $\mathcal{S}$ into minimum number of bins. For a given bin size, there always exists an optimal solution for bin packing. Let the bin size be $1 - \epsilon$, and r_{min} be the minimum number of bins used for packing $\mathcal{S}$ in one of the optimal solutions. Further, let $\mathcal{B}in_1, \mathcal{B}in_2, \cdots, \mathcal{B}in_{r_{min}}$ be the r_{min} bins, $\mathcal{B}in_i = \{a_1, a_2, \cdots, a_{n_i}\}$, $v_i = \sum_{i=1}^{n_i} a_i \leq 1 - \epsilon$, and $u_i = 1 - v_i$, where $1 \leq i \leq r_{min}$. Let the above optimal solution defined by the mapping function, $f()$, which maps $\mathcal{S}$ to the

integers of 1 to r_{min}. For the schedule generated by H_k, let $\mathcal{B}_i$ and I_i be as defined in the previous section. Our proof basically adds the length of each I_i such that $\sum_{i=1}^{m} I_i$ reaches the maximum. The bound will depend on m, k and r. Specifically, we derive two different bounds, one for the case where $r \geq r_{min}$ and another for $r < r_{min}$. Lemmas 13, 14, 15 derive the bound for $r \geq r_{min}$. Lemmas 16, 17, and 18 derive the bound for $r < r_{min}$. Finally, theorem 5 proves the main result for uniform tasks.

Lemma 13: Under the uniform task model, any feasible schedule generated by the H_k algorithm has the following property:

$$\sum_{i=1}^{j} \frac{i}{j} I_i \leq L_O$$

where $j < k$.

Proof Assume that all tasks have unit computation time. Examining the number of tasks in $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_j$, where $2 \leq j < k$, we have

$$\sum_{i=1}^{j} i \cdot I_i \leq j \cdot L_O.$$

Otherwise, in the time interval sets $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_j$, the number of tasks is more than $j \cdot L_O$. By the *pigeonhole principle*, there must exist a subset of tasks with more than j tasks which can be executed in parallel. The H_k algorithm will find it and put it into one of the time interval sets $\mathcal{B}_{j+1}$, $\mathcal{B}_{j+2}$, $\cdots$, $\mathcal{B}_m$. It is a contradiction that this subset of tasks uses the time intervals belonging to $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_j$.

To complete the proof, the lemma is derived by rewriting the above inequality by dividing both sides by j. $\square$

The following lemma considers a case where the number of processors is relatively small compared to the number of resource types. The opposite case will be considered later.

Lemma 14: For uniform tasks,

$$\frac{L_{H_k}}{L_O} \leq \frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j}$$

where $2 \leq k \leq m$.

Proof By examining any schedule generated by H_k, the schedule is divided into the time interval sets: $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_m$, and uses I_1, I_2, $\cdots$, I_m to represent their lengths defined as before. We have

$$\sum_{i=1}^{k-1} i \cdot I_i + k \cdot \sum_{i=k}^{m} I_i \leq \sum_{i=1}^{k-1} i \cdot I_i + \sum_{i=k}^{m} i \cdot I_i \leq \sum_{i=1}^{m} i \cdot I_i \leq \sum_{i=1}^{n} c_i \leq m \cdot L_O$$

where n is the number of tasks, c_i is the computation time of task T_i, which is same for all tasks. Noticing that $\sum_{i=k}^{m} I_i = L_{H_k} - \sum_{i=1}^{k-1} I_i$, we have

$$(\sum_{i=1}^{k-1} i \cdot I_i) + k \cdot \sum_{i=k}^{m} I_i = (\sum_{i=1}^{k-1} i \cdot I_i) + k(L_{H_k} - \sum_{i=1}^{k-1} I_i) \leq m \cdot L_O.$$

Rewriting the above inequality:

$$k \cdot L_{H_k} - \sum_{i=1}^{k-1} (k-i) \cdot I_i \leq m \cdot L_O.$$

Further,

$$k \cdot L_{H_k} \leq m \cdot L_O + \sum_{i=1}^{k-1} (k-i) \cdot I_i.$$

We first analyze the term $\sum_{i=1}^{k-1} (k-i) \cdot I_i$:

$$\sum_{i=1}^{k-1} (k-i) \cdot I_i = \sum_{i=1}^{k-1} i \cdot k \cdot (\frac{1}{i} - \frac{1}{k}) \cdot I_i$$

Since

$$(\frac{1}{i} - \frac{1}{k}) = \sum_{j=i}^{k-1} \frac{1}{j \cdot (j+1)},$$

$$\begin{aligned}
\sum_{i=1}^{k-1} i \cdot k \cdot (\frac{1}{i} - \frac{1}{k}) \cdot I_i &= \sum_{i=1}^{k-1} \sum_{j=i}^{k-1} \frac{i \cdot k}{j \cdot (j+1)} \cdot I_i \\
&= \sum_{j=1}^{k-1} \sum_{i=1}^{j} \frac{i \cdot k}{j \cdot (j+1)} \cdot I_i \\
&= \sum_{j=1}^{k-1} \frac{k}{j+1} \cdot (\sum_{i=1}^{j} (\frac{i}{j}) \cdot I_i), \qquad \text{by lemma 13}
\end{aligned}$$

$$\leq \sum_{j=1}^{k-1} \frac{k}{j+1} \cdot L_O.$$

So,

$$k \cdot L_{H_k} \leq m \cdot L_O + \sum_{j=1}^{k-1} \frac{k}{j+1} \cdot L_O$$

By rewriting the above inequality and changing the index bound, the lemma is derived. □

The next lemma says that there exists an example which shows the above bound is reachable.

Lemma 15: For given m and k, where $2 \leq k \leq m$, if $r \geq r_{min}$, there exists a case x, which is a particular processor-resource-task parameter setting, such that,

$$\frac{L_{H_k}(x)}{L_O(x)} = \frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j}.$$

Proof If we find such x, the lemma is proven.

For any given m, k, and r which satisfy the conditions of the lemma, we can always find n tasks with particular resource requests and deadlines such that the above equation becomes true.

Define $d = LCM(2, 3, \cdots, k)$, where $LCM()$ is the least common multiple function. There are $d \cdot m$ tasks with a unit computation time.

$$\mathcal{T}_6 = \{T_{1,1}, T_{1,2}, \cdots, T_{1,d}; T_{2,1}, T_{2,2}, \cdots, T_{2,d}; \cdots; T_{m,1}, T_{m,2}, \cdots, T_{m,d}\}.$$

Since $r \geq r_{min}$, $\mathcal{T}_6$ can only use r_{min} types of resources. They are $R_1, R_2, \cdots, R_{r_{min}}$ (i.e. each resource corresponds to a bin defined above). For simplicity here we assume that each task can ask for a fraction of a resource. (This corresponds to the fraction of the number instances of a resource type required by a task.)

Each $T_{i,j}$ requests a_i of $R_{f(a_i)}$ for exclusive use and requests u_l of R_l for shared use, where $l \neq f(a_i)$, $1 \leq l \leq r_{min}$, $1 \leq i \leq m$, and $1 \leq j \leq d$.

We force the H_k algorithm to schedule the task set $\mathcal{T}_6$ in the following order by adjusting their deadlines:

$$(T_{1,1}, T_{1,2}, \cdots, T_{1,d}; T_{2,1}, T_{2,2}, \cdots, T_{2,d}; \cdots; T_{m,1}, T_{m,2}, \cdots, T_{m,d}).$$

The optimal algorithm uses the following order:

$$(T_{1,1}, T_{2,1}, \cdots, T_{m,1}; T_{1,2}, T_{2,2}, \cdots, T_{m,2}; \cdots; T_{1,d}, T_{2,d}, \cdots, T_{m,d}).$$

Then it can be verified that the length of the schedule generated by the H_k algorithm is:

$$L_{H_k} = \frac{(m-k+1)\cdot d}{k} + \frac{d}{k-1} + \frac{d}{k-2} + \cdots + \frac{d}{2}$$

and the length of the schedule generated by the optimal algorithm is $L_O = d$. So the bound is reached:

$$\begin{aligned} \frac{L_{H_k}}{L_O} &= \frac{m-k+1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + \cdots + \frac{1}{2} \\ &= \frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j}. \end{aligned}$$

□

The above two lemmas (14 & 15) show that there exists a tight bound, $(m/k + 1/2 + 1/3 + \cdots + 1/k)$, if $r \geq r_{min}$. In what follows, we derive another tight bound for the case of $r < r_{min}$.

First, let us consider an extreme case of scheduling tasks with resource requirements when m is unbounded. The scheduling problem is immediately mapped to a multi-dimensional bin packing problem where each time unit in a schedule becomes a bin and the minimizing the schedule length becomes minimizing the number of bins.

Lemma 16: (M. R. Garey, R. L. Graham, D. S. Johnson, and A. C.-C. Yao [4])
With an unbounded m,

$$L_{List} \leq (r + \frac{7}{10}) \cdot L_O + \frac{5}{2}.$$

For uniform tasks, when we compare list scheduling with H_k, we find that H_k is just as greedy as list scheduling, because for any schedule generated by H_k, there is no single task that can be moved earlier without causing resource or processor conflicts. So, if m is unbounded, the bound of the above theorem is still true for the H_k scheduling algorithm under the uniform task model. So we have:

Lemma 17: With an unbounded m,

$$L_{H_k} \leq (r + \frac{7}{10}) \cdot L_O + \frac{5}{2}.$$

We now use the above result to handle the case of bounded m.

Lemma 18:

$$L_{H_k} \leq (r + \frac{17}{10} - \frac{(w+1) \cdot (r + \frac{7}{10} - \sum_{i=1}^{w} \frac{1}{i})}{m} - \frac{w}{m}) \cdot L_O + \frac{5}{2}$$

where $2 \leq k \leq m$ and w is the largest possible value such that $w < k$ and

$$\sum_{i=1}^{w} \frac{L_O}{i} \leq (r + \frac{7}{10}) \cdot L_O + \frac{5}{2}.$$

Proof For any schedule of length L_{H_k}, $\mathcal{B}_m$ accounts for "processor bounded" time intervals with a total length I_m. So we have

$$\begin{aligned}
L_{H_k} &= \sum_{i=1}^{m-1} I_i + I_m \\
&\leq \sum_{i=1}^{m-1} I_i + \frac{m \cdot L_O - \sum_{i=1}^{m-1} i \cdot I_i}{m} \\
&\leq \sum_{i=1}^{m-1} I_i + L_O - \sum_{i=1}^{w} \frac{i}{m} \cdot I_i - \frac{w+1}{m} \cdot \sum_{i=w+1}^{m-1} I_i \\
&= \sum_{i=1}^{m-1} I_i + L_O - \sum_{i=1}^{w} \frac{i}{m} \cdot I_i - \frac{w+1}{m} \cdot \sum_{i=1}^{m-1} I_i + \frac{w+1}{m} \cdot \sum_{i=1}^{w} I_i \\
&= (1 - \frac{w+1}{m}) \cdot (\sum_{i=1}^{m-1} I_i) + L_O + \frac{w+1}{m} \cdot \sum_{i=1}^{w} I_i - \frac{w}{m} \cdot \sum_{i=1}^{w} \frac{i}{w} \cdot I_i.
\end{aligned}$$

Let us analyze the terms in the above inequality. The time interval sets, $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_{m-1}$, account for "resource bounded" time intervals in the schedule generated by H_k, and their total length is unchanged if there are more processors being added. So, by the above corollary,

$$\sum_{i=1}^{m-1} I_i \leq (r + \frac{7}{10}) \cdot L_O + \frac{5}{2}.$$

For the term $\sum_{i=1}^{w} I_i$, we have

$$\sum_{i=1}^{w} I_i \le \sum_{i=1}^{w} \frac{1}{i} \cdot L_O,$$

where $w < k$, based on the following reasons: By lemma 13, we have

$$\sum_{i=1}^{w} i \cdot I_i \le w \cdot L_O.$$

For all the tasks in $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_w$, applying H_w algorithm on w processors, we have

$$\sum_{i=1}^{w} I_i = L_{H_w}.$$

Then apply lemma 14:

$$L_{H_w} \le \sum_{i=1}^{w} \frac{1}{i} \cdot L_O. \qquad \square$$

For the term $\sum_{i=1}^{w} \frac{i}{w} \cdot I_i$, we can directly apply lemma 13.

So,

$$\begin{aligned}
L_{H_k} &\le (1 - \frac{w+1}{m}) \cdot ((r + \frac{7}{10}) \cdot L_O + \frac{5}{2}) + L_O + \frac{w+1}{m} \cdot \sum_{i=1}^{w} \frac{L_O}{i} - \frac{w}{m} \cdot L_O \\
&= (r + \frac{17}{10} - \frac{(w+1) \cdot (r + \frac{7}{10} - \sum_{i=1}^{w} \frac{1}{i})}{m} - \frac{w}{m}) \cdot L_O + \frac{5}{2} \cdot (1 - \frac{w+1}{m}) \\
&\le (r + \frac{17}{10} - \frac{(w+1) \cdot (r + \frac{7}{10} - \sum_{i=1}^{w} \frac{1}{i})}{m} - \frac{w}{m}) \cdot L_O + \frac{5}{2}.
\end{aligned}$$

$\square$

We now show that this is a general result, e.g., when $k = 2$, $r = 1$ and $w = 1$, the above lemma degenerates to the following corollary.

Corollary 1: (K. L. Krause, V. Y. Shen, and H. D. Schwetman [7])

$$L_{List} \le (\frac{27}{10} - \frac{24}{10 \cdot m}) \cdot L_O + 2.$$

Notice that there is a small difference in the last term when the lemma degenerates to the above corollary, and the last term is a small constant and can be ignored when L_O is large.

[7] has shown that the bound of the above corollary can be reached asymptotically, that means that for $k = 2$ the bound of the above lemma can be reached asymptotically also. For $k \geq 2$, we can use a similar method to show that the bound of the above lemma can be reached asymptotically.

By combining lemma 14, 15, and 18, we have the following theorem.

Theorem 5: The H_k algorithm has a tight schedule length bound for the uniform task model,

$$\frac{L_{H_k}}{L_O} \leq min\left\{\frac{m}{k} + \sum_{j=2}^{k}\frac{1}{j}, \quad r + \frac{17}{10} - \frac{(w+1)\cdot(r + \frac{7}{10} - \sum_{i=1}^{w}\frac{1}{i})}{m} - \frac{w}{m}\right\}$$

where $2 \leq k \leq m$ and w is the largest possible value such that $w < k$ and

$$\sum_{i=1}^{w}\frac{L_O}{i} \leq (r + \frac{7}{10})\cdot L_O + \frac{5}{2}.$$

For any given m, the above theorem gives us a sequence of bounds for different values of k. So, if a scheduler is willing to spend more time, a better bound can be guaranteed. In one extreme, for a relatively large r, H_m algorithm gives us a bound:

$$\frac{L_{H_m}}{L_O} \leq \sum_{j=1}^{m}\frac{1}{j} \approx \ln\, m + \gamma$$

where $\gamma = 0.57721\cdots$ is *Euler's constant.*

The same idea can be used to generalize list scheduling. List scheduling has the schedule length bound of $(m + 1)/2$. In the schedule generated by a list scheduling algorithm, it keeps at least two processors busy except at the end of the schedule. Call this version of list scheduling, L_2. If we generalize L_2 to L_k, such that L_k attempts to keep at least k processors busy if possible, otherwise it works as list scheduling. L_k is useful in a non real-time environment. By the above theorem, we have a corollary:

Corollary 2: The L_k algorithm has a tight schedule length bound for the uniform task model,

$$\frac{L_{L_k}}{L_O} \leq min\left\{\frac{m}{k} + \sum_{j=2}^{k}\frac{1}{j}, \quad r + \frac{17}{10} - \frac{(w+1)\cdot(r + \frac{7}{10} - \sum_{i=1}^{w}\frac{1}{i})}{m} - \frac{w}{m}\right\}$$

where $2 \leq k \leq m$ and w is the largest possible value such that $w < k$ and

$$\sum_{i=1}^{w} \frac{L_O}{i} \leq (r + \frac{7}{10}) \cdot L_O + \frac{5}{2}.$$

For both the H_k algorithm and the L_k algorithm, it is highly desirable to measure their ability for finding feasible schedules.

7 Conclusions

In this paper we discuss the performance of approximation algorithms for hard real-time scheduling. In some cases both the ability to generate feasible schedules and the quality of the generated feasible schedules, expressed in terms of the schedule length bound, are important criteria. We analyzed the schedule length bound of the heuristic algorithm H_1 and have shown that it is equal to m. We identified those aspects of the heuristic algorithm which cause the bound to be m and developed the H_2 algorithm which has good performance with respect to both criteria. We analyzed the schedule length bound for the H_2 algorithm and shown that it is equal to $(m+1)/2$, that is, almost half that of H_1. Simulation results show that the H_2 algorithm has almost the same mean behavior when it comes to finding feasible schedules. Finally, we generalized the heuristic scheduling algorithm into the H_k algorithm, one that attempts to keep k processors busy. We analyze the schedule length bound for the generalized algorithm. For tasks with same computation time, we show that the bound is the minimum of

$$\left\{ \frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j}, \quad r + \frac{17}{10} - \frac{(w+1) \cdot (r + \frac{7}{10} - \sum_{i=1}^{w} \frac{1}{i})}{m} - \frac{w}{m} \right\}$$

where $2 \leq k \leq m$ and w is the largest possible value such that $w < k$ and

$$\sum_{i=1}^{w} \frac{L_O}{i} \leq (r + \frac{7}{10}) \cdot L_O + \frac{5}{2}.$$

For tasks with arbitrary computation time, we show the bound is

$$\frac{L_{H_k}}{L_O} \leq min \left\{ \frac{m+1}{2}, r + 1 + \frac{m - 2r - 1}{k} \right\},$$

where $2 \leq k \leq m$. In both case, the complexity is $O(n^{k+1}r^k)$, where $2 \leq k \leq m$, and n is the number of tasks. We also generalize the traditional list scheduling algorithm along the same lines and show the result of its analysis.

References

[1] J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[2] Jr. E. G. Coffman, editor. *Computer and Job-Shop Scheduling Theory.* John Wiley & Sons, 1976.

[3] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. on Computing*, 4:187–200, 1975.

[4] M. R. Garey, R. L. Graham, D. S. Johnson, and A. C.-C. Yao. Resource constrained scheduling as generalized bin packing. *J. Combinatorial Theory Ser. A*, 21:257–298, 1976.

[5] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. on Computing*, 4:397–411, 1975.

[6] Peter L. Hammer, editor. *Scheduling under Resource Constraints – Deterministic Models*, volume 7. J. C. Baltzer AG, 1986.

[7] K. L. Krause, V. Y. Shen, and H. D. Schwetman. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *J. ACM*, 22:522–550, 1975.

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

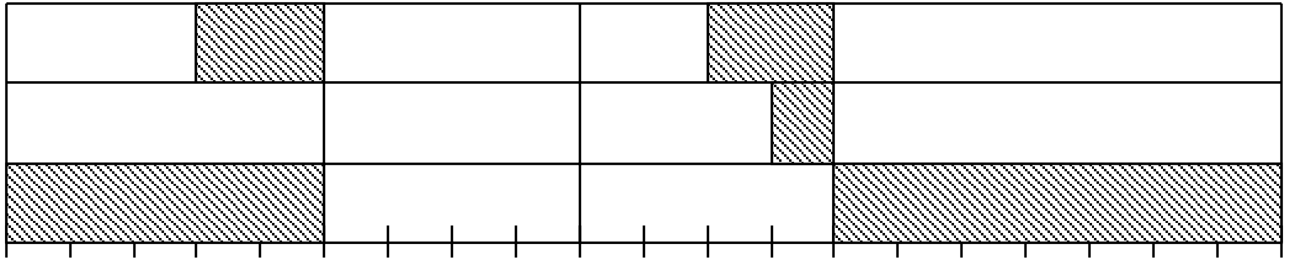
[9] K. Ramamritham, J. A. Stankovic, and P. Shiah. O(n) scheduling algorithms for real-time multiprocessor systems. *Proceedings of the International Conference on Parallel Processing*, August 1989.

[10] K. Ramamritham, J. A. Stankovic, and P. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 1, April 1990.

[11] J. A. Stankovic and K. Ramamritham. The design of the spring kernel. *Real-Time System Symposium*, pages 146–57, December 1989.

[12] W. Zhao and K. Ramamritham. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *The Journal of System and Software*, 7:195–207, 1987.

[13]W. Zhao, K. Ramamritham, and J. A. Stankovic. Scheduling tasks with resource requirements in hard real-time system. *IEEE Trans. on Software Engineering*, SE-13(5), May 1987.



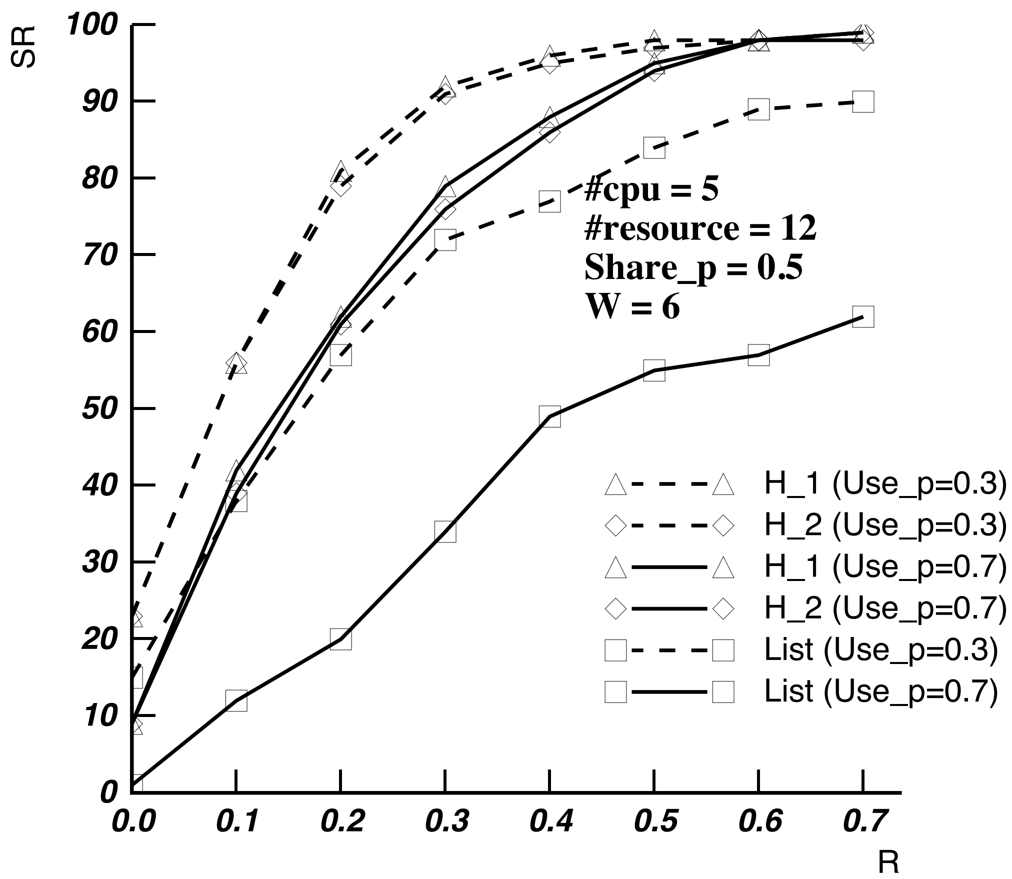


Fig. 2. Effect of R on three algorithms

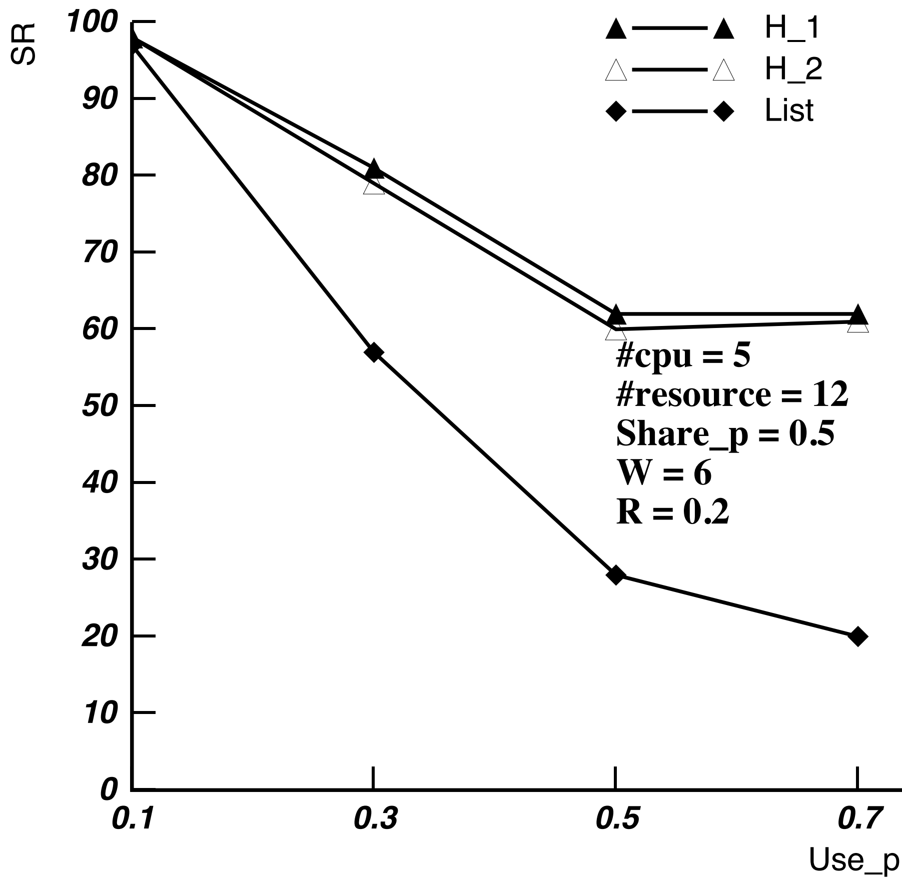


Fig. 3. Effect of resource contention on SR