Greed in Resource Scheduling

Donald W. Gillies and Jane W.-S. Liu

1304 W. Springfield Avenue
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

ABSTRACT

We examine the worst-case performance of a class of scheduling algorithms commonly known as priority-driven or list-scheduling algorithms. It is well known that these algorithms have anomalous, unpredictable performance when used to sub-optimally schedule nonpreemptive tasks with precedence constraints. We present a general method for deriving the worst-case performance of these algorithms. This method is easy to use, yet powerful enough to yield tight performance bounds for many classes of scheduling problems. We demonstrate the method for several problems to show it has wide applicability. We also present several task systems for which list-scheduling algorithms have worst-case performance and discuss the general characteristics of these task systems. We believe that these task systems are sometimes ignored in simulation studies; consequently, the results of these studies may be overly optimistic.

1. Introduction

It is important in real-time systems to minimize the unpredictability in the completion time of every real-time task. Unpredictability arises when two or more tasks contend for shared resources, such as processors, memory pages, or communication channels. Tasks that compete for shared resources may block until other tasks holding the resources release them. The length of time a task stays blocked depends on the resource requirements and execution times of the other tasks, making it hard to predict the task completion time. One way to eliminate this unpredictable behavior is to allocate dedicated resources to every task, but this is expensive and in some applications impossible. Another way to eliminate this unpredictability is to dynamically pre-allocate resources, so that when a task begins execution, it is guaranteed to have the resources it needs for the duration of its execution. This is accomplished by solving the resource contention problem together with the task scheduling problem. These combined problems are called real-time resource scheduling problems.

This work supported in part by a contract from the ONR (NVY-N00014-89-J1181).

Unfortunately, most resource scheduling problems encountered in practice are NP-hard; yet fast algorithms are typically needed. We are concerned with a class of simple heuristic algorithms that never leave resources idle intentionally. These algorithms are known as *priority-driven* or *list-scheduling algorithms.* Whenever sufficient resources are available, these algorithms schedule the ready task with the highest priority according to a priority list. The priority of a task may be assigned statically or dynamically; commonly used criteria for priority assignment include the task's programmer-defined priority, relative criticalness, deadline, execution time, or resource requirements [1,5,7,8,15]. Because they try to make the best local choice at each scheduling decision point, list-scheduling algorithms are also called greedy algorithms. They are particularly suited to real-time scheduling because they can often be implemented to run in time $O(n \log n)$.

As illustrated by Graham's well-known examples [7], a list schedule, produced by a list-scheduling algorithm, may increase in length when task execution times decrease, when more resources are made available, or when the task dependency graph is relaxed. Therefore, list-scheduling algorithms generate a new source of unpredictability. This paper addresses the problem of bounding the worst-case performance of these algorithms, and hence, bounding this new unpredictability.

To bound this unpredictability we find the maximum length of a aub-optimal list schedule using linear programming. This method is powerful enough to yield tight performance bounds for many classes of problems, such as those analyzed in [6,7,9,10,11,12]. We illustrate the method by working several sample problems. The performance bounds we obtain allow us to determine the schedulability of a task system according to some list-scheduling algorithm. We present several task systems which cause list-scheduling algorithms to exhibit their worst-case performance and discuss the general characteristics of these tasks systems.

The remainder of this paper is organized as follows: Section 2 describes our assumptions about the scheduling problems considered here and introduces the terminology used in later sections. Section 3 outlines the linear-programming method of analysis. Section 4 works four sample problems and presents task systems for which the worst-case bounds are tight. We also plot performance curves to aid in understanding the bounds. Section 5 discusses greedy algorithms in general, presents several lower bounds for independent task systems (no precedence constraints), and partially characterizes the problems that can be solved optimally by greedy algorithms. Section 6 summarizes our results.

2. Assumptions

Resources come in many types. For instance, message buffers, processors, communication channels, database locks, and memory pages can all be thought of as resources. The resources we consider in this paper have four qualities.

1. *Resources are renewable* - Renewable resources are a standard assumption in scheduling problems. In other words, the resources are not consumed as they are scheduled. (For information on scheduling with nonrenewable resources, see [3].) Hence, resources assigned to a task can be reassigned to other tasks when the task completes.

2. *Resources are nonpreemptable* - We consider only the case of nonpreemptive tasks. Resources are assigned to a task when it is scheduled and are unavailable until it completes.

3. *Resources need not be contiguously allocated* - This implies that memory must be managed by compaction or page-based allocation. This assumption reduces the complexity of the scheduling problem. Other researchers have considered the problem of scheduling resources that must be contiguously allocated [4,13].

4. *Resources come in fixed integer quantities* - Our models are discrete. Each type of resource is available in an integer amount, and tasks request integer amounts of one or several resource types. We further assume that for every resource type, there are upper and lower bounds on the number of resources requested by any task. This allows us to model processors as resources, by defining a separate type of resource as "processors" and setting the upper and lower bounds to one, if every task needs exactly one processor. If tasks need several processors, then the upper bound may be greater than one.

All the scheduling problems considered here are extensions of the following problem: We are given a system of tasks $\mathbf{T} = \{ T_1, T_2, ..., T_N \}$, and every task has ready time equal to zero. We wish to schedule these tasks in a system with r identical resources to minimize the overall completion time of all tasks in $\mathbf{T}$. Each task requires $R(T_i)$ resources for $\mu(T_i)$ units of time. We refer to $R(T_i)$ as the *width* of T_i and $\mu(T_i)$ as the *length* of T_i. Among all tasks, let $k = \max_{\text{all } i} \{R(T_i)\}$ be the maximum width and $l = \min_{\text{all } i} \{R(T_i)\}$ be the minimum width. There is also a partial order $<$ defined over $\mathbf{T}$. If $T_i < T_j$, then T_i must complete before T_j may begin, and T_i is a *predecessor* of T_j. A task is *ready* when all its predecessors have completed. A schedule produced by a list-scheduling algorithm is known as a *list schedule*, and the time at which all tasks in $\mathbf{T}$ are complete is the *length* of the schedule.

An example of a task system is depicted in Figure 1a. Individual tasks are represented by the vertices in a directed graph. There is an edge from T_i to T_j if $T_i < T_j$. Tasks are labeled by their (*name*, *length*, *width*), so (T_5, ε, 4) indicates that task T_5 requires 4 resources for ε units of time. Task sizes are approximated with ovals. Figure 1b shows a list schedule of the task system in Figure 1a. The length of this schedule is 8. For simplicity, we will omit axis divisions and labels in future schedules.

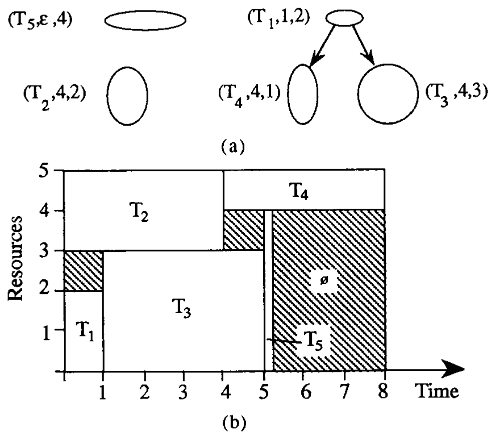


Figure 1. Sample Task System

In many problems there are multiple resource types. When discussing these problems, we will index each type by a number, and each variable will be qualified by the subscript j. Thus, a task T_i requires $R_j(T_i)$ resources of type j, from a total of r_j type-j resources, and the maximum and minimum widths are k_j and l_j respectively. In the special case when the resources are processors and every task requires only one processor, we say that there are m processors, and the maximum and minimum widths are understood to be one.

Again, we want to find the worst-case performance of list-scheduling algorithms. Let W be the length of an optimal schedule, and W' be the length of any list schedule. We observe that the time in any schedule may be divided into two disjoint parts. During *busy time* all the resources are fully utilized. On the other hand, during *idle time*, at least one resource is not fully utilized, so that a task could conceivably start in this time. By bounding the lengths of these times, we will obtain an upper bound on W'/W which will often be tight.

3. A Method for Deriving Worst-Case Performance

We will use linear-programming [14] to derive the worst-case performance of list-scheduling algorithms. In a linear programming problem we are given a set of s linear inequalities (known as *constraints*), in n variables, of the form:

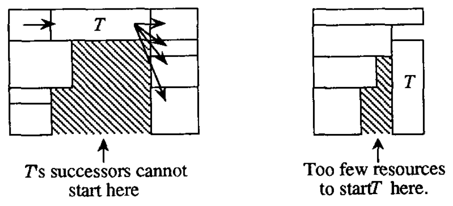
$$a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + \dots + a_{in}x_n \quad \leq \quad C_i$$

where $1 \leq i \leq s$ and the a_{ij} and C_i are all constants. In our problems the x_i's denote the lengths of different types of idle times, depending on the problem. We will give more details later. The problems we solve in this paper will have no more than 6 constraints, but much larger problems can be solved by commercial software packages. There will be one constraint equation for each type of resource, and several simple constraints to make all the x_i's non-negative. Each equation partitions n-dimensional space into a half that satisfies the inequality and one that does not. The intersection of all the satisfying halves is known as a *simplex*, which can be thought of as a geometric shape in n dimensions having flat faces and sharp corners. We are also given a linear equation (called the *objective function*) of the form:

$$y \; = \; b_1x_1 + b_2x_2 + b_3x_3 + \dots + b_nx_n$$

The problem is to find a set of x_i within the simplex that maximizes the objective function. For our problems, $b_i = 1$ for all i. The objective function will tell us the length of a worst-case schedule, and each x_i will be the length of the corresponding idle time in such a schedule. A simple brute-force algorithm to solve a linear programming problem is to convert all the inequalities to equalities, solve them simultaneously to obtain the simplex corner points, and take the point that maximizes the objective function. The point in space that maximizes the sum of the idle time corresponds to the worst-case performance.

As we stated earlier, all time in a schedule can be divided into busy time and idle time. The length of the busy time is denoted by W_b, and the intervals of time during which all resources are busy is referred to as W_b-time. The idle time can further be divided into many types. Sometimes resources are left idle because precedence constraints prevent the remaining tasks from being ready. This time is called the *idle time due to precedence constraints*, and the total duration is denoted by W_p. At other times, some tasks are ready but cannot begin because there are not enough free resources of type j. This time is called the *idle time due to resource constraints*, and the total duration is denoted by W_r, or by W_{r_j} when there are several types of resource. When two or more types of resources are unavailable in sufficient quantities to start any task, we arbitrarily attribute this time to the resource type of least index j. We speak of the W_p-time and the W_r-time as the intervals of time when resources are idle because of precedence or resource constraints, respectively. The two forms of idle time are depicted in Figure 2. In [7], it was shown that the W_p-time lasts for no more than W time units, the length of an optimal schedule. This is because there is a chain of tasks executing during W_p-time, and this chain is no longer than the length of an optimal schedule. Hence, $W_p \leq W$.


T
T's successors cannot start here
T
Too few resources to start T here.

(a) W_p-time (b) W_r-time

Figure 2. Different Types of Idle Time

The following method is used to derive the worst-case performance of list-scheduling nonpreemptive tasks with precedence constraints.

STEP 1: Define a variable for each type of idle time in a worst-case schedule.

STEP 2: For each type of resource, write a linear inequality relating idle-time resource utilization rates to the length and the maximum resource utilization of an optimal schedule.

STEP 3: Add the following constraints:
(a) Each idle time variable is non-negative.
(b) The duration of W_p-time is less than W ($W_p \leq W$).

STEP 4: Solve the linear program to maximize the sum of all the types of idle time. This is the maximum length of a list schedule.

STEP 5: Construct a worst-case schedule, employing the solution from step 4, to show the bound is tight.

Some care is required to apply step 2 of this method in order to get a tight bound. In an optimal schedule, tasks utilize at most r_j type-j resources for at most W units of time. A worst-case schedule is divided into disjoint types of time. If during these times type-j resources are utilized at some constant minimum rates, then a linear constraint can be written to bound the type-j resource usage in a worst-case schedule by the usage in an optimal schedule. After applying the method a task system should be constructed to verify that this minimum utilization rate can be maintained. We will illustrate these steps in the next section.

4. Sample Problems

In this section we solve four scheduling problems to illustrate our method. First, we solve the simplest version of our problem, where just m processors are to be scheduled. Second, we extend the problem to the scheduling of r resources where the task widths may vary. Then, we use the method to get tight bounds on the combined scheduling of m processors and r resources. This problem originally motivated us to develop the method. It is a simple extension to add several resource types to this problem; however, a closed form solution is too complex to derive. Finally, we consider an intricate problem involving two resources that are used in several ways, by two kinds of tasks. This problem stretches the method to its limit and generates $O(r)$ variables, where r is the amount of resource available.

4.1 Scheduling Simple Tasks

We consider scheduling m processors alone, and use our method to obtain the bound of $2\text{-}1/m$, an early result in scheduling theory [7]. The only type of idle time in this problem is the W_p-time. During this time, at least one processor is executing. During W_b-time, m processors are executing. Therefore, we have the following inequalities from steps 1-3.

$$\begin{aligned} 1\cdot W_p + m\cdot W_b &\le mW && \text{(1a)} \\ W_b &\ge 0 \\ W_p &\ge 0 \\ W_p &\le W && \text{(1b)} \end{aligned}$$

Processors are the only resource in this problem. The left-hand side of (1a) is the sum of the resources in use times the lengths of the idle times in a list schedule. In other words, it is a lower bound on the work done on a task system by the processors in a worst-case schedule. The right hand side is an upper bound on the work done on a task system by the processors in an optimal schedule. The rest of the equations are standard. By inspection of (1a), we see that W_p+W_b is maximized when W_p is as large as possible; hence $W_p = W$, a limiting value from (1b). From (1a) we get $W_b \le (m - 1)W/m$, so $W' = W_p+W_b \le (2 - 1/m)W$.

4.2 Scheduling Resources of a Single Type

In this problem we are given r resources and a task system **T**. Task widths may vary in the range $l \le R(T_i) \le k$. This problem models computations on massively parallel machines where each task executes on many processors at once.

During the W_p-time, at least one task of width l uses resources. During W_r-time, at most $k - 1$ resources are idle; otherwise a ready task could start, and such a task always exists during W_r-time. Therefore, at least $r - (k - 1)$ resources are in use during this time. This bound is simple but not as tight a possible. What follows is a better bound that results in a tight bound when all the tasks have a common width greater than one.

Let $g = \underset{all\ i}{gcd}\{R(T_i)\}$ be the greatest common divisor of all the task widths. At any time the number of resources in use must be an integer multiple of g. Let $\lceil x \rceil_b = b \cdot \lceil \frac{x}{b} \rceil$ denote the least integer multiple of b that is larger than or equal to x. Note that $\lceil x \rceil_1 = \lceil x \rceil$.

Lemma 1. *During the W_r-time, the maximum amount of free (idle) resource is $\phi_r = r - \lceil r - (k - 1) \rceil_g$.*

Proof. During W_r-time, at most $k - 1$ resources are idle, otherwise any ready task may be scheduled and such a task always exists during W_r-time. However, $r - (k - 1)$ may not be a multiple of g; it may be impossible for $k - 1$ resources to be idle. In fact, the least multiple of g that exceeds $r - (k - 1)$ is $\lceil r - (k - 1) \rceil_g$ □

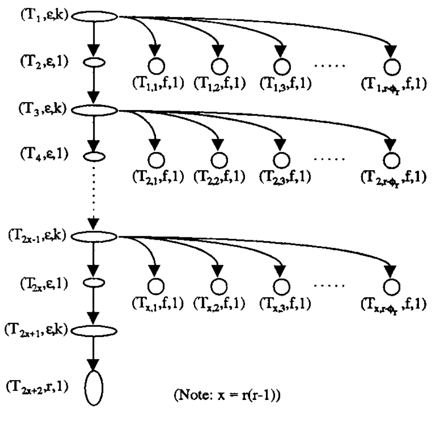
If all the tasks have a width that is a multiple of some $g > 1$, then at most $(r - r \bmod g)$ resources can be in use at any time in an optimal schedule. Thus, the linear equations are

$$\begin{aligned} l\cdot W_p + (r - \phi_r)\cdot W_r &\le W(r - r \bmod g) && \text{(2a)} \\ W_p &\ge 0 \\ W_r &\ge 0 \\ W_p &\le W \end{aligned}$$

We observe that $\phi_r \le r - l$, because $l \le k$. By inspecting (2a) we see that $W_p + W_r$ is maximized when $W_p = W$. Hence (2a) gives a value for W_r, $W_p = W$, and we have

$$\frac{W'}{W} = \frac{W_p + W_r}{W} \le 2 + \frac{\phi_r - l - r \bmod g}{r - \phi_r} \qquad \text{(2b)}$$

The example shown in Figures 3-5 demonstrates that this upper bound is tight. This example is complex because it handles all possible values of r and k simultaneously. We assume that $g = 1$. (An example for $g > 1$ can be obtained from our example by multiplying all the task widths and the number of resources by g and adding $r \bmod g$ resources.) In this example, $l = 1$. When $l > 1$, we may widen all the unit-width tasks to width l, if l divides r. We do not consider the case when l does not divide r.



(T_1,ε,k)
$(T_2,\varepsilon,1)$
$(T_{1,1},f,1)$ $(T_{1,2},f,1)$ $(T_{1,3},f,1)$ ..... $(T_{1,r-\phi_r},f,1)$
(T_3,ε,k)
$(T_4,\varepsilon,1)$
$(T_{2,1},f,1)$ $(T_{2,2},f,1)$ $(T_{2,3},f,1)$ ..... $(T_{2,r-\phi_r},f,1)$
(T_{2x-1},ε,k)
$(T_{2x},\varepsilon,1)$
$(T_{x,1},f,1)$ $(T_{x,2},f,1)$ $(T_{x,3},f,1)$ ..... $(T_{x,r-\phi_r},f,1)$
(T_{2x+1},ε,k)
$(T_{2x+2},r,1)$
(Note: $x = r(r-1)$)

Figure 3. Tasks for a System with r Resources

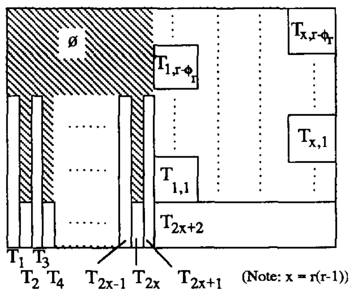


Figure 4. An Optimal Schedule

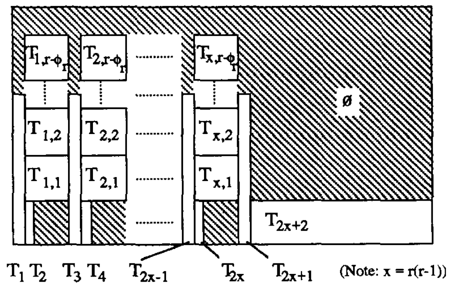


Figure 5. A Worst-Case Schedule

Figure 3 depicts the precedence graph of a worst-case task system. Figure 4 shows an optimal schedule of length $(2r(r-1)+1)\varepsilon + r$. This is not a list schedule because resources are intentionally left idle for $r(r-1)\varepsilon$ units of time at the beginning of the schedule. Figure 5 depicts a worst-case list schedule. The overall performance ratio for these two schedules is

$$\frac{W'}{W} = \frac{\varepsilon(r(r-1)+1) + r(r-1)/(r-\phi_r) + r}{\varepsilon(2r(r-1)+1)+r}$$

$$= \frac{(\varepsilon r(r-1)+1)(r-\phi_r) + r(2r-1-\phi_r)}{(\varepsilon(2r(r-1)+1)+r)(r-\phi_r)} \rightarrow 2 + \frac{\phi_r - 1}{r-\phi_r} \quad \text{as } \varepsilon \rightarrow 0$$

Thus the upper bound (2b) is tight. The schedule in Figure 5 is dictated entirely by the greedy nature of list scheduling; each task must start at the indicated time, because resources are free and only certain tasks are ready. We call this type of task system *perfectly tight*, because its list schedule cannot be shortened by a better priority list. Hence, *no priority assignment can avoid this worst-case performance bound.* Figure 6 plots the worst-case performance for tasks systems where the task width varies in the range $1 \le R(T_i) \le k$, and k is expressed as some percentage of r. If all tasks use less than half the resources, list scheduling is guaranteed to produce schedules that are within three times the length of an optimal schedule.

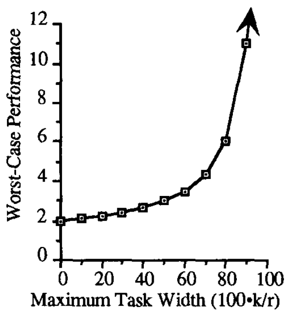


Figure 6. Asymptotic Scheduling Performance for One Resource

4.3 Scheduling Processors and Resources of a Single Type

Consider the list-scheduling of tasks in a system with m processors and r resources. Every task T_i requires one processor and $R(T_i)$ resources. The bounds k and l apply to the resources, and ϕ_r is the maximum number of idle resources during W_r-time, as given by Lemma 1. In this system a new type of idle time, called the W_m-time, occurs when all the processors are utilized. In other words, if more processors were available during W_m-time, then additional tasks could be scheduled in this time. Altogether there are three types of idle time in this system, W_p-time, W_r-time, and W_m-time. The linear inequalities are:

$$\begin{aligned} m \cdot W_m + \left\lceil \frac{r-\phi_r}{k} \right\rceil \cdot W_r + 1 \cdot W_p &\le m \cdot W && \text{(3a)} \\ ml \cdot W_m + (r - \phi_r) \cdot W_r + l \cdot W_p &\le (r - r \bmod g) \cdot W && \text{(3b)} \\ W_r &\ge 0 \\ W_m &\ge 0 \\ W_p &\ge 0 \\ W_p &\le W && \text{(3c)} \end{aligned}$$

The first inequality is written for the processors. It states that in a worst-case list schedule no more processor time is used than in an optimal schedule. In a worst-case schedule during W_m-time, m processors are in use; during W_r-time, at least $(r - \phi_r)$ resources are in use, and hence $\lceil (r-\phi_r)/k \rceil$ processors must be in use; and during W_p-time, at least one processor is in use. In an optimal schedule, at most m processors are in use for W time units.

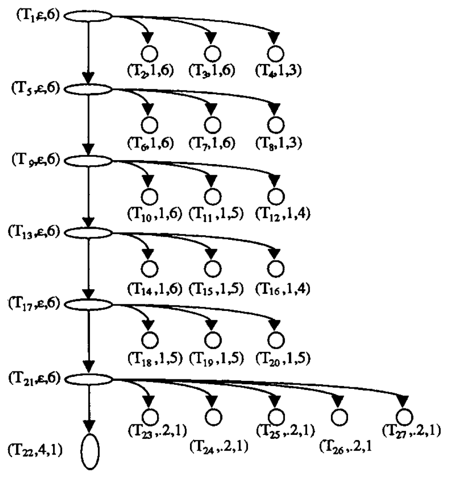
The second inequality is written for the resources. It states that in a worst-case list schedule no more resources are used than in an optimal schedule. During the W_m-time, at least $m\,l$ resources are in use; during the W_r-time, at least $(r - \phi_r)$ resources are in use; and during W_p-time, at least l resources are in use. In an optimal schedule, at most $r - r \bmod g$ resources are in use for W units of time.

These inequalities determine a simplex in 3 dimensions, with (non-axis) faces given by (3a) (3b) and (3c). Since the coefficients before W_p in both (3a) and (3b) are minimal among all coefficients, we conclude that the maximum value occurs when $W_p = W$. This will almost always happen in practice. By setting $W_r = 0$ or setting $W_m = 0$, we obtain bounds similar to those in Sections 4.1 and 4.2 since this problem is a generalization of both these problems. By converting (3a) and (3b) to equalities and solving for an intersection point on the surface of the simplex, we get a complicated formula. The formula and a task systems for all three possible solutions are presented in [6]. Here we only calculate the bound and present a task system for one value of the parameters r, m, l, k, and g.

As an example, we consider a system with m = 5 processors, r = 20 resources, and tasks that vary in width from l = 1 to k = 6. Hence ϕ_r = 5, g = 1, $r \bmod g$ = 0, and

$$\begin{aligned} 5W_m + 3W_r + W_p &\le 5W \\ 5W_m + 15W_r + W_p &\le 20W \end{aligned}$$

Setting $W_m = 0$ yields a bound of W'/W = 34/15 (2.2666...), and setting $W_r = 0$ yields W'/W = 2.25. The largest value of the three points is at the simplex intersection point, which yields W_r/W= .25, W_p/W = 1, and W_r/W = 1.05, hence W'/W = 2.30. Figure 7 shows a task system for which this bound is tight, to within some arbitrarily small ε. Tasks are numbered by their priorities, because the task system is not perfectly tight. Figure 8 gives one of several optimal schedules for this task system. Processors are not depicted, but it is evident that no more than 5 tasks are running at once. Figure 9 shows a worst-case list schedule for this task system. The task system was constructed by decomposing an optimal schedule into pieces of appropriate sizes, given by W_m/W, W_r/W, and W_p/W.



(T1,ε,6)
(T2,1,6) (T3,1,6) (T4,1,3)
(T5,ε,6)
(T6,1,6) (T7,1,6) (T8,1,3)
(T9,ε,6)
(T10,1,6) (T11,1,5) (T12,1,4)
(T13,ε,6)
(T14,1,6) (T15,1,5) (T16,1,4)
(T17,ε,6)
(T18,1,5) (T19,1,5) (T20,1,5)
(T21,ε,6)
(T22,4,1)
(T23,.2,1) (T24,.2,1) (T25,.2,1) (T26,.2,1 (T27,.2,1)

Figure 7. Tasks for a System with 5 processors and 20 resources.

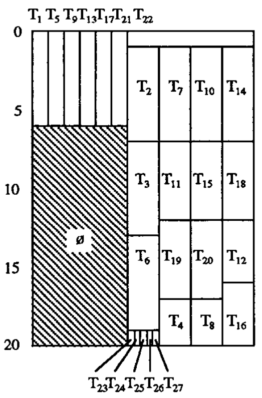


Figure 8. An Optimal Schedule

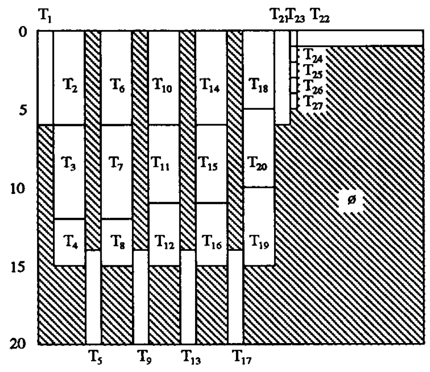
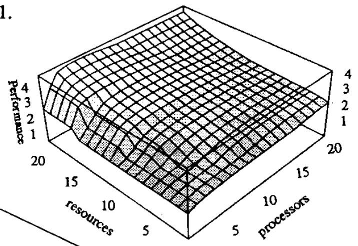


Figure 9. A Worst-Case Schedule

In [6] a closed-form performance bound was derived. Figure 10 and 11 depict the performance plots, with $l = g = 1$, for r = 20 and r = 102 respectively. Both plots begin at 2 resources since the performance bound is degenerate for one resource and one processor. The second plot is truncated for $k \ge 92$, since the curve climbs rapidly to 102 in this range, dwarfing the rest of the figure. Asymptotic performance resembling Figure 6 appears at the outside edge of Figure 11.

Figure 10. Performance with 20 Resources



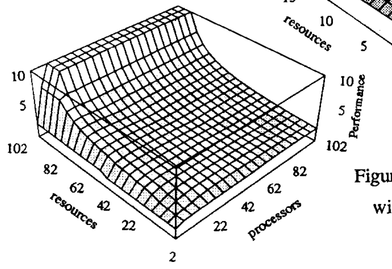


Figure 11. Performance with 102 Resources

4.4 Scheduling Processors, Resources, and a Matrix facility

In this problem we are again given m processors and r resources. However, two kinds of tasks exist in this system. A type-1 task requires 1 processor and $R(T_i)$ resources, as in the previous problem. A type-2 task makes use of a parallel matrix-manipulation facility. This facility requires $m/2$ processors to be available. A type-2 task must have its own processor to issue requests to the facility. Furthermore, up to q type-2 tasks may use the facility at once.

In this system W_P-time, W_r-time and W_m-time exist, as in the previous problem. However, we divide the W_p-time into W_{p_i}-time, where $i = 1$ or $i = 2$, and i is type of task in the chain that executes during W_p-time. We also divide the W_m-time into several types. First, there is W_m-time, as in the previous problem, when all processors are occupied by type-1 tasks. Then for $1 \le h \le q$ there is a new type of W_m-time that occurs when h type-2 tasks occupy the matrix manipulation facility and h processors, and $m/2 - h$ type-1 tasks occupy the rest of the processors. This time is denoted by W_{m_h}. There is also idle time that occurs when the matrix manipulation facility is completely utilized by q tasks; this is called W_q-time.

As a final constraint, suppose we know that in an optimal schedule, no more than a fraction f of the processing power of the matrix facility is used. We have the following inequalities from steps 1-3.

$$(r - \phi_r)\cdot W_r + l \cdot W_{p_1} + \sum_{h=1}^{q} (\frac{m}{2} - h)\cdot W_{m_h}\cdot l \le (r - r \bmod g)\cdot W \quad (4a)$$

$$\left\lceil \frac{r - \phi_r}{k} \right\rceil \cdot W_r + 1\cdot W_{p_1} + (1 + \frac{m}{2q})\cdot W_{p_2} + \sum_{h=1}^{q} (\frac{m}{2} + \frac{mi}{2q})\cdot W_{m_h} + m\cdot W_m \le m\cdot W \quad (4b)$$

$$\frac{m}{2q}\cdot W_{p_2} + \sum_{h=1}^{q} (\frac{mh}{2q})\cdot W_{m_h} + m\cdot W_m \le f\cdot\frac{m}{2}\cdot W \quad (4c)$$

(4a) ensures that in a worst-case schedule the task system uses no more resources than in an optimal schedule. (4b) ensures that in a worst-case schedule, the task system uses no more processor time than in an optimal schedule. (4c) ensures that in a worst-case schedule, the task system spends no more time using the matrix manipulation facility than it does in an optimal schedule. In these inequalities, the use of the matrix manipulation facility is modeled as a request to use $\frac{m}{2q}$ processors. All three inequalities have $O(q)$ variables, because of the W_{m_h}-time. Therefore, it is impractical to solve this problem analytically, but performance bounds may still be computed using a linear programming software package. This example demonstrates that the solution method can handle complex problems.

5. Performance Limitations of Greedy Algorithms

List-scheduling, that is, greedy algorithms are commonly used in real-time scheduling. In this section, we investigate some inherent performance limitations of greedy algorithms. We want to answer the question, "Which problems can be optimally solved using a greedy algorithm?" For these problems greedy algorithms can obtain better performance by spending more time to find a better priority list. Some problems that can be optimally solved by a greedy algorithm include scheduling independent tasks on a multiprocessor, most preemptive scheduling problems, scheduling dependent unit-length tasks on 2 processors, and scheduling unit-length tasks with integer release times, integer deadlines, and precedence constraints, on a single processor.

Unfortunately, in several problems optimal schedules contain intentionally inserted idle time, which cannot be generated by greedy algorithms. These problems include scheduling unit-length tasks with *real-valued* release times and deadlines and all the problems considered in Section 4. The problem of scheduling independent tasks in a system with a single resource (as in Section 4.2, but independent tasks) belongs to this class, as we now show.

This problem is similar to some 2-dimensional (2-D) bin-packing problems. In 2-D bin packing, independent rectangles are packed into a bin of width r and unlimited height. The goal is to minimize the height of the packing. It is evident that the problem of scheduling independent tasks on r resources can be formulated as a 2-D bin packing problem. However, the problems are not identical; resource scheduling is easier than 2-D bin packing because tasks need not occupy contiguous resources. One of the first algorithms proposed for 2-D bin packing was the greedy bottom-left algorithm, which packs each rectangle as low as possible, then slides it leftwards as much as possible. Brown has shown in [2] that no bottom-left 2-D bin packing algorithm can guarantee a packing that is within 5/4 times the height of an optimal packing. The optimal bin packing in Figure 12, which is the unique packing of height 4 and width 7 (except for symmetry), was used to obtain this result.

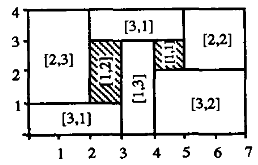


Figure 12. Worst-Case Example for Bottom-Left 2-D Bin Packing

In the appendix, we consider a bottom-left/right algorithm, which may choose to slide each block left or right after packing it as low as possible. We show that this algorithm cannot do better than 6/5 times the length of an optimal schedule.

We have noticed that Brown's example provides a lower bound for our scheduling problem. Consider resource scheduling the tasks in Figure 12 sideways on 4 resources, from left to right. These tasks may be scheduled non-contiguously. It is shown in the appendix that in all schedules of length 7 the given tasks start at the same times as in Figure 12 (except for symmetry). If we shorten both shaded tasks by ε, the opposite of widening them in Brown's proof, the optimal schedule still has length 7, although now there are two slices of idle time of length ε and width 1 and 2, respectively. Suppose that a greedy scheduler produces this schedule, from either end. Regardless of whether the schedule is produced from left-to-right or right-to-left, one or two resources are intentionally idle for ε time units (after a shaded task completes), while a [3,1] task is ready. Hence, no greedy algorithm can produce this schedule. Therefore, as $\varepsilon \rightarrow 0$, no greedy algorithm for independent tasks can produce resource schedules shorter than 8/7 times the length of an optimal schedule.

The remainder of this section gives hints about how to identify scheduling problems that require inserted idle time. For such problems, there is a lower bound that is perfectly tight, and no greedy algorithm can be improved beyond a certain point. In the following, we decompose a greedy algorithm is into two steps: the priority-assignment step, which may take arbitrarily long, and the control step, which removes tasks from the priority list, schedules them, and runs in polynomial time.

Theorem 1. If a scheduling problem has complexity outside NP, then no greedy algorithm can always produce optimal schedules.

Proof. Suppose that there is a scheduling problem with complexity outside NP, which can always be solved optimally by a greedy algorithm. Then clearly, for every solution, there exists a priority list that leads to the necessary solution. If such a priority list always exists, then it can be guessed in nondeterministic polynomial time. This implies the greedy algorithm is in NP, a contradiction. □

Thus, if a problem has complexity outside NP, then any greedy algorithm has a lower bound that is perfectly tight. This theorem is useless for many of the above example problems because they are in NP. We now give a rule-of-thumb for determining when greedy algorithms can produce optimal schedules if given unlimited time to produce an optimal priority list. When a greedy algorithm runs it gradually schedules tasks from a set. At any time, some unscheduled tasks in the set may not be available for scheduling, because there are insufficient processors or resources, or because a precedence constraint has not been satisfied, or for other reasons. We call a *decision point* a point in time in a greedy schedule when a task becomes available for scheduling. In a preemptive problem, a greedy algorithm always has the option of stopping all the running tasks at a decision point.

Heuristic: If a task can monopolize any processor or resource beyond the time of the next decision point, then the greedy algorithm *may not* be optimal.

This implies that greedy algorithms are optimal for most types of preemptive scheduling, since all tasks may be stopped whenever a decision point occurs. It also separates the two problems of scheduling unit-length tasks with release-times and deadlines, under integer-valued and real-valued inputs. This rule also lead us to discover the 8/7 lower bound for resource scheduling.

When empirically evaluating greedy algorithms, it is important to know if a problem can be optimally solved by a greedy algorithm. If the problem cannot be optimally solved, then problems requiring intentionally-inserted idle time should be included in the evaluation set. Otherwise, the empirical results could be overly optimistic, leading the researcher to draw the wrong conclusion about the performance of the greedy algorithm.

6. Conclusion

This paper shows how to analyze the worst-case behavior of a large class of scheduling algorithms applied to nonpreemptive tasks with precedence constraints, using a simple linear programming method. It demonstrates lower bounds that are *perfectly tight*, i.e. they could not be improved by devising clever priority lists. It partially characterizes the class of problems that are amenable to optimal solution by greedy algorithms. This characterization helps in two ways. First, it gives the algorithm designer a hint about when he can trade off more computation time for a better greedy solution. Without this knowledge, he might waste time attempting to improve a greedy algorithm when, in fact, such improvements are impossible. Second, it tells the empirical evaluation team when to include task systems requiring inserted idle time in their test cases.

In the future we plan to compare greedy resource-scheduling algorithms to *optimal greedy algorithms*, not optimal algorithms themselves. Many bin packing algorithms such as First-Fit-Decreasing and Next-Fit-Decreasing provide good performance on independent tasks. We are searching for a way of modeling these algorithms using linear programming. Finally, we are looking for a more precise characterization of the scheduling problems that can be optimally solved by greedy algorithms.

7. References

[1] Biyabani, Sara R., John A. Stankovic and Krithi Ramamritham. The Integration of Deadline and

Criticalness in Hard Real-Time Scheduling. *Proceedings of the IEEE Real Time Systems Symposium* (1988) pp. 152-160.

[2] Brown, Donna J. An Improved BL Lower Bound. *Information Processing Letters* (29 August 1980) vol. 11, no. 1, pp. 37-39.

[3] Carlier, J. and A. H. G. Rinnooy-Kan. Scheduling Subject to Nonrenewable-Resource Constraints. *Operations Research Letters* (1982) vol. 1, pp. 52-55.

[4] Coffman, E. G., M. R. Garey, D. S. Johnson and R. E. Tarjan. Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms. *SIAM Journal on Computing* (1980) vol. 9, pp. 808-826.

[5] Coffman, E. G., Jr. and R. L. Graham. Optimal Scheduling for Two-Processor Systems. *Acta Informatica* (1972) vol. 1, pp. 200-213.

[6] Gillies, Donald W. and Jane W. S. Liu. Performance of List Scheduling with Resource Constraints. Manuscript (1989).

[7] Graham, R. L. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* (March 1969) vol. 17, no. 2, pp. 416-429.

[8] Liu, C. L. and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM* (January 1973) vol. 20, no. 1, pp. 46-61.

[9] Liu, J. W. S., and C. L. Liu. Performance Analysis of Multiprocessor Systems Containing Functionally Dedicated Processors. *Acta Informatica* (1978) vol. 10, pp. 95-104.

[10] Liu, J. W. S., and C. L. Liu. Bounds on Scheduling Algorithms for Heterogeneous Computing Systems. Technical Report UIUCDCS-R-74-632 (1974), Department of Computer Science, University of Illinois at Urbana-Champaign.

[11] Lloyd, E. L. Concurrent Task Systems. *Operations Research* (1981) vol. 29, pp. 189-201

[12] Lloyd, E. L. List Scheduling Bounds for UET Systems with Resources. *Information Processing Letters* (12 February 1980) vol. 10, no.1, pp. 28-31

[13] Sleator, Daniel D. A 2.5 Times Optimal Algorithm for Packing in Two Dimensions. *Information Processing Letters* (12 February 1980) vol. 10, no. 1, pp. 37-40.

[14] Strang, Gilbert. *Linear Algebra With Applications.* Academic Press, New York, 1980.

[15] Zhao, Wei, Krithivasan Ramamritham and John A. Stankovic. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers* (August 1987) vol. 36, no. 8, pp 949-960.

Appendix 1.

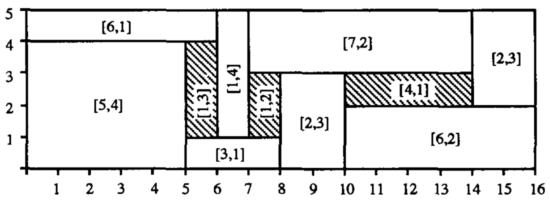


Figure A.1. Worst-Case Example for Bottom-L/R 2-D Bin Packing

Theorem 1 shows this is the unique packing (discounting flips) of width 16 and height 5. We now widen each shaded rectangle by ε, and widen the bin by 3ε. The optimal packing still has height 5 and looks the same. But to get an optimal packing the third row must be packed correctly. However, in any optimal packing, two separate idle-time gaps must exist at the bottom row of the packing, because the [1,4] or [2,3] block "hangs down" from the fully-occupied third row. Yet a greedy left/right heuristic cannot produce two non-adjacent gaps on the bottom row. Thus, this packing cannot be obtained. Therefore, no bottom-left/right heuristic can pack 2-D bins with a worst-case height less than 6/5 times the height of an optimal packing.

Theorem 1. *The packing in Figure A.1 of height 5 and width 16 is unique (discounting symmetry)*.

Proof. Consider row 3. Since blocks [2,3], [2,3], [1,4], [1,3] and [5,4] are all at lest 3 units tall, they are horizontally disjoint in a packing of height 5. Since their total width is 11, we cannot have [7,2], [6,2], or [6,1] overlap row 3, or else this row sum would be 17. Hence [7,2], [6,2], and [6,1] must be at the top/bottom or bottom/top of any packing. Along any of the 5 rows in a packing, there is a sequence of blocks with a total width of 16. Let (x,y,...) denote the the widths of individual blocks on a given row, totalling 16. The block widths will be reordered to appear in decreasing order.

Lemma 1. (6,6,x_1,..) cannot occur on any row

Proof. (6,6,5,...) is too wide for a single row, so if (6,6,...) occurs, then [6,1] is completely above/below [5,4]. Without loss of generality assume (6,6,...) occurs on row 5, with [5,4] resting below, as in the picture. Then we have:

case 1: (6,6,4) or (6,6,3,1) on row 5. Then because row 5 of the packing is filled, we must have [5,4], [7,2], [2,3], [2,3], [1,4] overlapping row 2, but the total width is 17.

case 2: (6,6,2,2) on row 5. Then [1,4] must be packed beneath [6,1]; recall [7,2] must be packed on the bottom. The remaining row space from row 5 to row 1 is {0,0,6,3,3}. [4,1] and [1,3] must be packed on rows 3 and 1, leaving space of {0,0,1,2,2}, which clearly cannot support the [3,1] block.

case 3: (6,6,2,1,1) on row 5. Because the leftover space is {0,1,x,x,x} row 1 contains (7,5,2,..) and hence [7,2] [5,4] [2,3]. However, this sequence cannot end with a 2 (impossible; the other [2,3] block is at the top), or a 1,1 (impossible, 2 of 3 blocks of width 1 have been used at the top). Hence the row 1 sum cannot be 16.

Hence [6,2] rests on the bottom of the packing, and [7,2] rests at the top (discounting flips). Even after this simplification, there are 23 row sequences totalling 16, so the problem is not easily solved using the algebraic equation method in [2]. The free space at this point is {3,4,11,5,5}.

Lemma 2. There must be 3 separate rows with (...,6,...,2,...)

Proof. We establish this by showing there is a [2,3] completely below [7,2] (resting at the top of a packing), and a [2,3] completely above [6,2] (resting at the bottom of a packing).

Case 1: Assume no [2,3] is completely below [7,2]. Then on row 4 the following blocks are horizontally disjoint: [5,4] [7,2] [2,3] [2,3] [1,4], and the horizontal sum is 17. Assume both [2,3]'s are completely below [7,2], then (depending on [1,4]) we have either free space of {3,3,6,0,0} (can't accommodate [4,1] [3,1] and [1,3] in this space) or {2,3,6,0,1} (impossible to pack row 1). Hence exactly one [2,3] is completely below [7,2]

Case 2: Assume no [2,3] is completely above [6,2]. From the previous case we conclude one [2,3] must rest on row 1 of the packing, and the other must rest on row 2 for the theorem to be false. This leaves {3,2,7,1,3} in free space since both [2,3]'s are determined This free space indicates that packing the [1,4] block will completely fill row 2, hence [1,4] must rest on row 2 to allow row 1 to be fully packed with the [3,1] block leaving free space of {2,1,6,0,0}. But then remaining [1,2] and [1,3] blocks cannot both be packed because of row 4.

Theorem 3. No packing (up to isomorphism) besides the given one exists.

Proof. We sight along the rows, having established the base rows of [5,4] [1,6] [7,2] [6,2] and both [2,3]'s to be the same as in the figure. The free space is {1,2,7,3,3}. Block [4,1] must be on row 3, giving (1,2,3,3,3). Block [1,3] completely fills row 1, 2, or 3; to also accommodate block [1,4] it must rest on row 1 row; subtracting both yields {0,1,2,2,0} in free space. Therefore, blocks [1,2] and [1,3] must rest on the second row.

This establishes that in all valid packings blocks rest on rows given in the picture. We consider sliding blocks along their rows to obtain a different packing. Considering [5,4] and [6,1]. we find they must be neighbors in any packing. By flipping them once or twice, we obtain the same relationship as in the figure. The following sequence of grid points (actually, blocks below & right of these points) inductively determines the packing in Figure A.1: (1,4); (6,1); (7,2); (8,2); (8,5); (16,5); (16,2); (14,2).

Theorem 3. *The following packing is the unique packing of width 4 and length 7:*

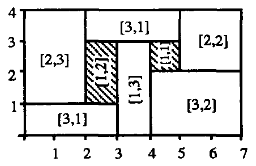


Figure A.2 A Schedule for a System of Independent tasks

Proof. Consider the resources in a schedule of length 7 decomposed into strips of width 4 and length 1. If all the resources in a strip are busy, then this can happen in three possible ways a, b, c, depending on the widths of the tasks that fill a strip.

$$a = (2,2)$$
$$b = (3,1)$$
$$c = (2,1,1)$$

The following equations ensure the number of tasks of width 1, 2, and 3 are the same as the number of tasks available in the task system in Figure A.2

$$2c+b = 7 \qquad \text{(1's equation)}$$
$$2a+c = 6 \qquad \text{(2's equation)}$$
$$b = 3 \qquad \text{(3's equation)}$$

Hence the unique solution is $a = c = 2$, b=3. Since $c = 2$, both [3,1] tasks cannot start at the same time, otherwise we would have $c = 3$. If they start one time unit apart, no matter when the [1,1] task executes, we have either $c > 2$ or $b > 2$. The [3,1] tasks must start no more than two time units apart, otherwise $c \leq 1$. This and b implicitly determine relative starting times for the [3,1], [3,1], and [2,3] tasks, except for vertical or horizontal flips, to look like the picture Consider that $a = 2$ and some task of width 2 must execute during the overlap between the two [3,1] tasks, and the fact that [1,3] and [3,2] must execute at disjoint times in a schedule of length 7. This determines the relative starting time for the [1,2] task, and a determines the relative starting times between the [2,2] and [3,2] tasks, except for horizontal flips. Thus we have determined the relative starting times of the [2,2] & [3,2] tasks, the [2,3] & [1,3] & [1,3] & [1,3] tasks -- the [1,3] and [1,1] tasks are yet to be determined. Now the values of a, b, and c, or simple exhaustive enumeration, suffice to show that any schedule of length 7 has tasks starting at the same times as in Figure A.2.