Journal of Computers and Operations Research, **Vol. 22, No. 10, pp. 1089-1100, Dec. 1995.**

Heterogeneous-Criteria Scheduling: Minimizing Weighted Number of Tardy Jobs and Weighted Completion Time[1]

Jon M. Peha
Carnegie Mellon University, Dept. of ECE, Pittsburgh, PA 15213-3890, USA
Phone: (412) 268-7126, Fax: (412) 268-2860, Email: peha@@ece.cmu.edu

Scope and Purpose

The optimal scheduling algorithms in this paper have many applications, but they were motivated by research into real-time systems and integrated-services networks. Consequently, two features of the algorithms are of particular importance: they must have low complexity, not just polynomial complexity, and to the extent possible, they should accommodate *diverse performance objectives*. This diversity takes two forms. First, it is more important that the scheduler achieve good performance for some jobs than others. Second, the performance measure that is appropriate for one job may not be appropriate for another job. Although there has been research into *multicriteria* scheduling algorithms which attempt to optimize the same measures of performance (or *criteria*) for all jobs, little work has been done for the case we call *heterogeneous-criteria scheduling* where jobs are divided into classes, and the performance measure differs from class to class. For application in real-time multiprocessor systems, a third property is important; algorithms should work on multiple machines. This paper presents novel low-complexity scheduling-algorithms that accommodate diverse performance objectives on one or more machines, including both single-criteria and heterogeneous-criteria problems.

Abstract

In this paper, we present a novel $O(N^2)$ algorithm to minimize the weighted number of tardy jobs with unit processing times, integer ready times and deadlines, and M homogeneous parallel machines, where N is the number of jobs to be scheduled. w_i is the weight reflecting job i's importance, and U_i is 1 if job i is tardy and 0 otherwise, so $\sum w_i U_i$ is the measure of performance. (In standard notation [1], this is the P/r_i, p_i=1/$\sum w_i U_i$ problem.) This algorithm is extended to minimize weighted completion time $\sum w_i C_i$ for some jobs, where completion time C_i is the time job i completes processing, and $\sum w_i U_i$ for other jobs, at the same complexity. (The P/r_i, p_i=1/$\sum w_i U_i$&$\sum w_i C_i$ problem.) Complexity can be reduced to $O(N\log N)$ if all ready times are the same (with one additional constraint on weights). (P/p_i=1/$\sum w_i U_i$&$\sum w_i C_i$.) Finally, if one is trying to find the number of tardy jobs of each weight under optimal scheduling, but an actual schedule is not needed, this (P/r_i,p_i = 1/$\sum w_i U_i$) problem can be solved with an $O(WN\log N)$ algorithm, where W is the number of different weight values. In some cases, this can be done analytically.

[1]This material is based upon work supported under Grant NCR-9210626 from the National Science Foundation.

Heterogeneous-Criteria Scheduling: Minimizing Weighted Number of Tardy Jobs and Weighted Completion Time

Jon M. Peha

1 Introduction

There has been a great deal of research into scheduling algorithms that optimize a single measure of performance for all jobs, such as the average job completion time or the average tardiness [1]. Although there has been much less research into *multicriteria* scheduling algorithms that instead optimize performance of all jobs according to multiple performance measures, this area is now growing [2, 3, 4]. However, there are important scheduling problems where a single performance measure, or even set of performance measures, is not appropriate for all jobs. Jobs are divided into multiple classes, and the appropriate measure of performance differs from one class of jobs to another. We call such problems *heterogeneous criteria* scheduling problems. For example, this occurs in integrated-services packet-switched networks such as ATM (asynchronous transfer mode) networks, and in soft real-time systems. In these systems, it is useful to classify jobs as either *time-constrained*, which should be processed any time before their deadlines, or *non-time-constrained*, which should simply be processed as early as possible. Regardless of the performance measure, it is also more important to achieve good performance for some jobs than others. Furthermore, in many important models that are relevant to the study of integrated-services networks and real-time systems, the number of jobs to be scheduled can be extremely large, so low complexity is essential. In this paper, we present novel low-complexity ($O(N^2)$ or better) approaches to the scheduling of constant-length time-constrained jobs, and the scheduling of constant-length *heterogeneous* job sets, which are sets that include both time-constrained and non-time-constrained jobs.

Associated with each job i is a ready time r_i, which is the earliest job i can begin processing, a processing time p_i, and a weight w_i that reflects job i's importance. If job i is time-constrained, it also has a deadline d_i. For non-time-constrained jobs, the objective is to minimize the weighted total completion time $\sum_{\forall i} w_i C_i$, where C_i is the time job i completes processing. For time-constrained jobs, it is the weighted number of tardy jobs $\sum_{\forall i} w_i U_i$ that should be minimized, where $U_i = 0$ if $C_i \leq d_i$, and $U_i = 1$ otherwise. When scheduling heterogeneous jobs, optimal performance is defined through hierarchical minimization, which is specified as follows. Let jobs be divided into sets $A_j : j \geq 1$, such that all of the jobs in a set are homogeneous, i.e. either all time-constrained or all non-time-constrained, and for any job i in A_j and any job k in A_{j+1}, $w_i > w_k$. (We restrict weights such that time-constrained and non-time-constrained jobs cannot have equal weights.) Let $P_j(S)$ be the performance of jobs in set j under schedule S, i.e. $P_j(S) = \sum_{i \in A_j} w_i C_i$ for a set of non-time-constrained jobs, or $P_j(S) = \sum_{i \in A_j} w_i U_i$ for a set of time-constrained jobs. The objective is to optimize performance for the more important sets, and to optimize performance of the less important sets to the extent possible without degrading performance of their more important counterparts. More precisely, the vector $\vec{P}(S)$ is the measure of performance to be minimized, where $\vec{P}(S_1) < \vec{P}(S_2)$ if $P_1(S_1) < P_1(S_2)$, or if $P_1(S_1) = P_1(S_2)$ and $P_2(S_1) < P_2(S_2)$, or in

general, if $P_j(S_1) = P_j(S_2)$ for all $j < k$ and $P_k(S_1) < P_k(S_2)$ for some k.

In order to find algorithms of polynomial complexity, constraints must be imposed on the general scheduling problem defined above. Even if all weights are equal, but r_i, d_i, and p_i can vary from job to job, scheduling would be NP-complete if all jobs are time-constrained [5], or if all jobs are non-time-constrained [1]. Consequently, we assume that $p_i = 1$ for all i. Even if p_i (and r_i) are constant for all i, the scheduling of time-constrained jobs is still NP-complete if w_i and d_i can vary [6], unless we also assume a slotted system, i.e. r_i and d_i are integers for all i. (These assumptions are appropriate for ATM networks, and reasonably approximate other systems.) With these assumptions, we will show that low-complexity scheduling is possible even for heterogeneous jobs.

In the next section, we describe the relevance of this scheduling problem to research in integrated-services networks and real-time systems. Section 3 presents two approaches for time-constrained jobs, or in standard notation [1], $1/r_i$, p_i=$1/\sum w_iU_i$. In Section 4, we extend the scheduling algorithm from Section 3.1 to accommodate non-time-constrained jobs as well as time-constrained jobs. This is $1/r_i$, p_i=$1/\sum w_iU_i\&\sum w_iC_i$, where we define the performance objective $\gamma_1\&\gamma_2$ in heterogeneous scheduling as minimizing γ_1 for some jobs and γ_2 for others. Section 5 presents a less complex scheduling algorithm for the special case where r_i is the same for all i, i.e. $1/p_i$=$1/\sum w_iU_i\&\sum w_iC_i$, under the constraint that only one set A_j contains time-constrained jobs. Finally, the paper is summarized in Section 6, and the results are extended to M homogeneous parallel machines, i.e. P/r_i, p_i=$1/\sum w_iU_i\&\sum w_iC_i$ and P/p_i=$1/\sum w_iU_i\&\sum w_iC_i$.

2 Relevance to Networks and Real-Time Systems

This paper was motivated by our research into integrated-services packet-switched networks such as the emerging ATM networks, and subsequently, from research in real-time systems. Both are discussed below. In any packet-switched network, information carried by the network is first divided into smaller pieces called packets. Packets are queued in a buffer at the network access point awaiting transmission into the network, and a scheduling algorithm orders these packet transmissions. Integrated-services networks are networks that carry diverse traffic types such as voice, video, image transfer, and various kinds of computer data, and diverse traffic implies diverse performance objectives. For example, for most types of computer data, performance is typically measured in mean queueing delay. Minimizing mean weighted queueing delay $\sum_{i=1}^{N} w_i(C_i - p_i - r_i)/N$ is equivalent to minimizing $\sum w_iC_i$, since only C_i depends on the scheduling. For voice and video, however, packets that are queued too long will not reach their destination in time for playback, and will be lost. Minimizing the weighted fraction that are lost $\sum_{i=1}^{N} w_iU_i/N$ is equivalent to minimizing $\sum w_iU_i$. Thus, in the transmission of packets in integrated-services networks, there are both time-constrained and non-time-constrained jobs. Since it is critical that voice and video applications consistently experience low loss rates, it is often the case that time-constrained jobs have greater weights, although this is not necessarily always true. Furthermore, not all jobs (packets) are equally important. For example, the loss of some voice or video packets are far more noticeable to a user than the loss of others, and it is more important to reduce the mean queueing delay of network control packets than the mean queueing delay of electronic mail. Thus, unequal weights are required.

The simple, traditional scheduling algorithms used in this context are first-comes-first-served, in

which jobs are served in the order in which they arrive, static priority, in which jobs are assigned priorities (weights) from a fixed range and the queued job with the greatest priority is served next, and earliest deadline first, in which the queued job with the earliest deadline is served next. However, by adopting a more sophisticated scheduling algorithm, much better performance can be achieved, or equivalently, the same performance can be achieved at a higher load [7, 8]. Consequently, in this context, several scheduling algorithms have been suggested in which the network designer can define an arbitrary number of traffic classes, and the appropriate measure of performance and weight can vary from class to class. These scheduling algorithms then discriminate based on each packet's performance measure, weight, and the delay it has already experienced. Such algorithms include *Cost-Based Scheduling* [9, 10], *Maximum Utility Scheduling for Transmission* [11], and most recently, the *Priority Token Bank* [12]. Many other possible approaches such as [13, 14] have been proposed for less general conditions, such as when there are a small number of traffic classes, and specific constraints on the performance objectives of each class, but the fact that some packets are time-constrained and others are not still affects scheduling decisions.

The same problems also arise when scheduling the execution of processes in a soft real-time system. A real-time system is a computer system in which the value of executing a process depends on when it is completed. For example, a process may be useful only if it completes before its deadline. In a *hard* real-time system, it must be guaranteed a priori that all deadlines will be met. In a *soft* real-time system, this is not possible, so the objective is to come as close to this ideal as possible. As in networks, jobs can be time-constrained or non-time-constrained, and some jobs are more important than others. Since some real-time systems (including the object of our research) can run on single or multiprocessor systems, it is also important that the scheduling algorithms extend to multiple machines.

When proposing a novel scheduling algorithm for integrated-services networks or real-time systems, it is often difficult to determine whether the algorithm is truly effective. Typically, one compares the new algorithm with the standard alternatives: first-come-first-served, static priority, and earliest deadline first. However, it is not difficult to outperform these simple algorithms. Our goal in developing an optimal algorithm is to enable more meaningful comparisons. The optimal algorithms described in this paper are assumed to have complete knowledge of future arrivals, so they always make the correct decision, yielding a valuable bound on achievable performance. For example, in [9, 10], we were able to demonstrate the effectiveness of the proposed Cost-Based Scheduling algorithm by comparing its performance with the optimum. Similar comparisons are made in [7], which seeks to show the limitations of the commonly used algorithms.[2] To make a meaningful comparison of expected long-term network performance, the algorithms must be applied to a large number of packets, so reducing the optimal algorithm's complexity is essential for keeping computation time practical.

Because the performance of time-constrained jobs is critical, the algorithms in Section 3 are useful in their own right. Because jobs are actually heterogeneous in networks and real-time systems, Section 4's enhancement makes the algorithm more valuable. There are also important uses for the

[2]This paper goes beyond its aforementioned predecessors by fully specifying the optimal algorithms, and then providing formal proofs of optimality. This paper also presents algorithms for scenarios not employed in [9, 10, 7] which have proved valuable in our research. These include the $O(WN\log N)$ approach for $1/r_i$, $p_i = 1/\sum w_i U_i$ in Section 3, the $O(N\log N)$ algorithm for $1/p_i = 1/\sum w_i U_i \& \sum w_i C_i$ in Section 5, and the extension of all of the above to M machines in Section 6, where N is the number of jobs and W is the number of values for weights.

algorithm in Section 5 which assumes all ready times are equal. In some real-time systems, jobs arrive in bursts, so a scheduler can reasonably operate under the assumption that no additional time-constrained jobs will arrive before the last burst is served. Furthermore, some advocate greedy scheduling algorithms that operate under this assumption even when there is actually a steady stream of new arrivals. This approach has been proposed both for real-time systems [15, 16, 17] and integrated-services networks [11], although the author has argued against this approach [10]. The algorithms are extended to multiple machines in Section 6.

3 Scheduling Time-Constrained Jobs

An algorithm for minimizing $\sum w_i U_i$ for time-constrained jobs is known for the case where $p_i = 1$, and r_i and d_i are integers for all i. As shown in [18], by repeatedly applying Glover's algorithm for convex bipartite graphs [19], $\sum w_i U_i$ can be minimized with an $O(N^3)$ algorithm, where N is the number of jobs. In our models of integrated-services networks, it is not unusual to need an N on the order of one hundred million. This can take on the order of ten hours of computation time with our $O(N^2)$ approach. Excessive computation would make an $O(N^3)$ approach unusable. There is also no obvious way to extend the algorithm in [18] to accommodate non-time-constrained jobs (for which the objective is to minimize $\sum w_i C_i$) as well as time-constrained jobs. A less complex and more extensible algorithm is needed. We will present an $O(N^2)$ algorithm for time-constrained jobs in Section 3.1, which will be extended to heterogeneous jobs in Section 4. Section 3.2 describes an alternative method of calculating the optimal achievable performance of time-constrained jobs, without producing an actual schedule. This approach is more efficient in some cases.

3.1 An $O(N^2)$ Scheduling Algorithm for Time-Constrained Jobs

We now present an algorithm that will find a schedule in which $\sum w_i U_i$ is minimized. Only jobs that meet their deadlines are included in this schedule. (The other jobs can be scheduled in any period left idle.) Let A be the set of jobs that will meet their deadlines. Clearly A should contain as many jobs as possible, particularly of the more important jobs. We initially let $A = \emptyset$, and then attempt to add jobs in non-increasing order of weight. For a job x, if all of the jobs in $A + \{x\}$ can be served before their deadline, then x is added to A; otherwise, x is discarded. The theorem below proves that the resulting set A is optimal.

To prove that this approach is optimal, we must prove that once a job is added to A, it should never be removed. Similarly, once a job is discarded from A, it should never be revived. The latter clearly follows from the former; if it is not possible to add a given job to A without some job missing its deadline, and no jobs in A are ever removed, then it will never be possible to add the job. Our theorem will prove that jobs added to A should never be removed. Note that this is not true unless $p_i = 1$ for all i, nor is it even true with $p_i = 1$ unless ready times and deadlines are integers. For example, with two jobs in which $w_1 = 5$, $r_1 = .5$, $d_1 = 1.5$, $p_1 = 1$, and $w_2 = 4$, $r_2 = 0$, $d_2 = 1$, $p_2 = 1$, only job 1 would meet its deadline, i.e. $A = \{1\}$. After adding job 3 in which $w_3 = 3$, $r_3 = 1$, $d_3 = 2$, $p_3 = 1$, job 1 would no longer meets its deadline in the optimal schedule; $A = \{2,3\}$.

To facilitate the proof, we first define a useful function. Let there be two schedules S_1 and S_2 in which jobs with $p_i = 1 : \forall i$ begin at integer times. If job J is scheduled to begin at time

t in S_1, job $f^{(1)}(J)$ is scheduled to begin at time t in S_2. If nothing is scheduled in S_2 at that time, then $f^{(1)}(J)$ is idle. This is useful in a proof, because it must be possible to replace J with $f^{(1)}(J)$ in S_1 without violating ready time or deadline constraints. Similarly, it must be possible to replace $f^{(1)}(J)$ with J in S_2. More complex replacements are also possible if $f^{(1)}(J)$ is also scheduled at some time in S_1. $f^{(2)}(J) = f^{(1)}(f^{(1)}(J))$ is the job scheduled in S_2 at the same time that $f^{(1)}(J)$ is scheduled in S_1. $f^{(2)}(J)$ can now be replaced by J in S_2, and J by $f^{(2)}(J)$ in S_1. For example, $f^{(1)}(J)$ is replaced by $f^{(2)}(J)$ in S_1, and then J is replaced by $f^{(1)}(J)$ in S_1. In general, $f^{(i+1)}(J) = f^{(1)}(f^{(i)}(J)) : \forall i > 0$. Also, if $f^{(j)}(J) \in S_1 : \forall 1 \leq j < i$, then $f^{(i)}(J)$ can replace J in S_1, or J can replace $f^{(i)}(J)$ in S_2, for all $i > 0$. For consistency, $f^{(0)}(J)$ is defined to equal J.

Theorem 1: A finite set A contains jobs in which $p_i = 1$, $w_i > 0$, and r_i and d_i are integers for all i. An algorithm that minimizes $\sum w_i U_i$ of jobs in A produces a schedule S_1. (Jobs that do not meet deadlines are not included in the schedule, and no job is included twice.) Job x is not in A, and x's weight is less than or equal to the weights of every job in A. There exists a schedule S_2 that minimizes $\sum w_i U_i$ for the jobs in $A + \{x\}$, where each job in S_1 is also in S_2.

Proof of Theorem 1 by contradiction: Let S_2 be chosen to minimize $\sum w_i U_i$ of jobs in $A + \{x\}$. Of those schedules in which $\sum w_i U_i$ is minimized, as many of the jobs in S_1 as possible are included in S_2. Assume that there exists a job y in S_1 that is not in S_2. We will prove that there must then be an infinite number of jobs that are in both S_1 and S_2, which is not possible. It is known that $f^{(0)}(y) = y$ is in S_1. We will show that if $f^{(j)}(y)$ is in S_1, for all $j : 0 \leq j < k$, then $f^{(k)}(y)$ is also in S_1 for all $k > 0$.

The conceivable values of $f^{(k)}(y)$ are: (1) idle, (2) x, (3) $f^{(j)}(y)$ for some $j : 1 \leq i < k$, (4) a job $\neq x$ and $\neq f^{(j)}(y)$ for all $j : 1 \leq i < k$, that has a weight less than y's, (5) such a job with a weight greater than y's, and (6) such a job with a weight equal to y's. We now consider each possibility, showing that the first four are impossible, and the last two both imply that $f^{(k)}(y)$ is in S_1. (In the process, recall that $\sum w_i U_i$ was minimized in both S_1 and S_2.) (1) $f^{(k)}(y)$ cannot be idle, because if it were, $\sum w_i U_i$ in S_2 could be improved by scheduling y at that time. ($w_y > 0$). (2) Job $f^{(k)}(y)$ cannot be x, because if it were, x in S_2 could be replaced by y, thereby increasing the number of jobs in S_1 that are also in S_2 without degrading $\sum w_i U_i$. ($w_x \leq w_y$). (3) $f^{(k)}(y)$ cannot be $f^{(j)}(y)$ for some $j : 1 \leq i < k$, because if it were, the job would be duplicated in S_2. (4) $f^{(k)}(y)$ cannot have a lower weight than y, because if it did, $\sum w_i U_i$ in S_2 could be improved by replacing $f^{(k)}(y)$ with y. (5) If $f^{(k)}(y)$ has a greater weight than y, then job $f^{(k)}(y)$ must be in S_1. Otherwise, y could be replaced by $f^{(k)}(y)$ in S_1, thereby improving $\sum w_i U_i$ in S_1. (6) If $f^{(k)}(y)$ and y have equal weights, then job $f^{(k)}(y)$ must be in S_1. Otherwise, y could be replaced by $f^{(k)}(y)$ in S_1, thereby increasing the number of jobs in S_1 that are also in S_2 without changing $\sum w_i U_i$. Thus, the job $f^{(k)}(y)$ is in both S_1 and S_2, but not in the same time slot.

Since $y = f^{(0)}(y)$ is in S_1, this proves by induction that an infinite number of unique jobs are in both S_1 and S_2, which is not possible, since there are only a finite number of jobs in A. Thus, the theorem is proved.

A method is now needed to determine whether the jobs in $A + \{x\}$ can all be scheduled before their deadlines, and if so, to find the corresponding schedule. If a schedule exists in which all jobs

meet their deadline, the earliest-deadline-first algorithm will produce such a schedule [5, 20]. With this algorithm, the job i scheduled at any time t is the one with the earliest deadline d_i of those queued, where a job is queued at time t if it has arrived ($r_i \leq t$), it has not already begun processing by time t, and it has not missed its deadline ($d_i - p_i \geq t$). The following algorithm generates a schedule consistent with earliest-deadline-first scheduling.

First, an attempt is made to schedule the new job x at time r_x. If that time slot is idle, a new schedule has been found. If not, either job x or the job previously scheduled at time r_x is scheduled at time r_x, and the other must be scheduled at a later time. The job allowed to remain at r_x is the one with the earliest deadline. In the next iteration, an attempt is made to schedule the currently unscheduled job in the next time slot, at time $r_x + 1$. This procedure is continued with successive time slots until either an idle time slot is found or a time slot is reached in which neither of the jobs considered can be scheduled at a later time without missing their respective deadlines. In the former case, the current job is scheduled in the idle time slot. In the latter case, all jobs are returned to the times at which they were scheduled before job x was introduced, and job x is discarded. The algorithm can be further enhanced by using a stack to store proposed changes in the schedule that should be enacted if all jobs can be scheduled before their deadline. If and only if x can be scheduled, the changes stored in the stack are made. The algorithm is shown below.

```
Let t = r_x                                   [t] is the job scheduled at time t
Let end = d_x
Clear stack
Push x onto stack
While (t < end) and (slot t is not idle) do
      If (d_[t] > end) then
            Let end = d_[t]
            Push t onto stack
            Push [t] onto stack
      Let t = t + 1
If (slot t is idle) then
      Pop stack into j
      Schedule job j at time t
      While (stack is not empty) do
            Pop stack into t
            Pop stack into j
            Schedule job j at time t
```

It can be shown that this algorithm is $O(N^2)$ as follows. For each job i scheduled, a series of time slots in the existing schedule must be scanned, beginning with time r_x, and ending either when an idle time slot is found or it is determined that the job cannot meet its deadline. With N jobs, no more than N time slots can be scanned before finding one that is idle. Thus, the algorithm is, at worst, $O(N^2)$. Since presorting by weight is only $O(N \log N)$ [21], it does not increase this complexity. Moreover, although theoretical complexity is $O(N^2)$, complexity in practice is better. The number of time slots that must be scanned cannot exceed the number of time slots B in a *busy period*, where a busy period is any period that contains no idle slots, and is both preceded and succeeded by an idle time slot. Complexity in practice is therefore $O(BN)$. In a queueing system where jobs arrive according to a stochastic arrival process, the average length of a busy

period can be derived analytically from that arrival process, and is independent of N, yielding a linear complexity. In our simulations of integrated-services networks, average B was often in the hundreds or thousands, while N could be in the hundred millions.

As a simple example of what can be done with this algorithm, consider a CPU queue to which real-time tasks arrive according to a Poisson process. Weights are distributed exponentially with mean 1. The amount of queueing delay that a task can tolerate before the result is no longer useful is $9-3E$, where E is an exponentially distributed random variable with mean 1, conditioned on the fact that $9-3E>0$. Figure 1 shows mean weighted loss rate ($\sum_{i=1}^{N} w_i U_i/N$) versus load, enabling the comparison of the following algorithms: optimal (OPT), cost-based scheduling (CBS) [9, 10], first-come-first-served (FCFS), static priority (SP), and earliest deadline first (EDF).

Figure at end of paper.

Figure 1: Weighted loss rate, $\sum_{i=1}^{N} w_i U_i/N$, versus load.

3.2 Method of Calculating Performance for Time-Constrained Jobs

In this section, we present an efficient means of determining for each weight the number of time-constrained jobs that would miss their deadlines under optimal scheduling, without ever determining the actual schedule. If the number of possible values for weight is relatively small, this technique is more efficient than the $O(N^2)$ algorithm presented in the previous section. In addition, with this approach, it is sometimes possible to derive long-term expected loss rate $\sum_{i=1}^{N} U_i/N$ as a function of weight analytically if one knows the job arrival process. (The latter approach has proved useful for models of real-time systems and integrated-services networks.)

Let the job types be numbered from 1 to W in descending order of weight, where all jobs of the same type have the same weight. $w^{(i)}$ is the weight assigned to type i jobs, and $L^{(i)}$ is the number of type i jobs that miss their deadlines under optimal scheduling. Our goal is then to find $L^{(i)} : 1 \leq i \leq W$. We first present a method for determining $L^{(1)}$, and then a technique is presented for finding $L^{(j)}$ when given $L^{(i)} : \forall i < j$.

As seen in Section 3, a job with weight $< w$ is scheduled before its deadline only when doing so will not affect the loss rate of the jobs with weights $\geq w$. Thus, when determining the loss rates of jobs with weights $\geq w$, the presence of jobs with weights $< w$ can be ignored. In particular, the loss rate $L^{(1)}$ of type 1 jobs under optimal scheduling can be determined by calculating the loss rate that would be achieved when only such jobs are present. In this case, optimal scheduling achieves the same loss rate as the $O(N \log N)$ earliest deadline first algorithm (EDF) [5, 20], which can be determined analytically in some cases.

The method of finding $L^{(j)}$ from $L^{(i)} : \forall i < j$ is based on the following theorem.

> *Theorem 2:* In a system where $p_i = 1$ and r_i and d_i are integers for all i, if $\sum w_i U_i$ is minimized in a schedule spanning a finite period for any set of weights in which $w_i > 0 : \forall i$, then $\sum U_i$ is simultaneously minimized.

Proof of Theorem 2: S_1 is a schedule spanning a finite time period in which $\sum w_i U_i$ is minimized, and S_2 is a schedule in which $\sum U_i$ is minimized over the same period. Jobs that miss their deadlines are not included in S_1 or S_2. $I(S)$ is the amount of idle time in schedule S. Since reducing $\sum U_i$ reduces the amount of idle time, $I(S)$ is minimized when $\sum U_i$ is minimized. Thus, $I(S_2) \leq I(S_1)$. Let k be the number of time slots in S_1 in which the job being served (or the idle period) differs from that of the corresponding slot in S_2. Let S_2 be chosen such that, of all the schedules in which $\sum U_i$ is minimized, k is smallest. We will prove that $I(S_1) = I(S_2)$ by contradiction. Assume $I(S_1) > I(S_2)$. This means there is at least one time slot which is idle in S_1 and not in S_2. Choose one such time slot, and consider the job i that occupies the slot in S_2. In S_1, this job is either lost or it is scheduled at another time. If the job is lost in S_1, then $\sum w_i U_i$ could be decreased by scheduling job i in this time slot rather than discarding job i. This contradicts the assumption that $\sum w_i U_i$ was minimized in S_1. In the other alternative, job i is scheduled at another time. In this case, job i could be moved to this time slot. This would reduce k by 1, and not change U_i, thereby not changing $\sum U_i$. This violates the assumption that S_2 was selected to minimize k. Thus, by contradiction, $I(S_1) = I(S_2)$, and consequently $\sum U_i$ for S_1 and S_2 are equal. S_1 must therefore have the minimum possible $\sum U_i$.

To find $L^{(j)}$, assume $L^{(i)} : \forall i < j$ are known. As described above, the presence of type $i : i > j$ jobs can be ignored. By Theorem 2, the $\sum U_i$ achieved by optimally scheduling the jobs from every class $i : i \leq j$ is the same as the loss rate L achieved when using earliest deadline first (EDF) with the same jobs. Since loss is conserved, $L^{(j)} = L - \sum_{i=1}^{j-1} L^{(i)}$ Thus, all loss rates can be determined by invoking EDF once for each of the W possible weights, yielding a complexity of $O(WN\log N)$.

In some cases, long-term expected loss rate $\sum U_i/N$ with earliest-deadline first scheduling can be found analytically, particularly if $d_i - r_i$ is the same for all i. To show the utility of this approach, we consider one such case where loss rates with earliest-deadline-first have been derived [22]. A communications channel has 20 independent sources. Each source alternates between active and inactive periods, and the durations of both are distributed exponentially. When a source is active, it generates data packets at a constant data rate f, and the mean duration of an active period is 50 ms. (The mean duration of an inactive period is a function of load.) The channel bandwidth is 150 Mb/s. A packet is considered lost if it is not transmitted within 1.1 ms of its arrival. A burst of data, which is the data generated in one active period by a single source, is of type 1 with probability p, and type 2 with probability $1 - p$. The time required to get weighted loss rates $\sum w_i U_i/N$ with the $O(N^2)$ algorithm described in the previous section would be excessive with this model, because busy periods can be so large. However, we can find loss rates analytically. Figure 2 shows the maximum load that can be tolerated with optimal scheduling (OPT) and earliest deadline first (EDF) under the constraint that no more than .3% of type 1 packets and 5% of type 2 packets are lost as a function of p. Data rates of f = 256 kb/s, 2 Mb/s, and 140 Mb/s are considered, yielding mean burst lengths of 12.8 kb, 100 kb, and 7 Mb, respectively. Thus, given the data rate and traffic mix, one can determine the effective capacity with the various algorithms.

Figure at end of paper.

Figure 2: Maximum load where requirements are met versus fraction of Class 1 packets, p, with traffic from 20 independent bursty sources.

4 An O(N^2) Algorithm for Heterogeneous Jobs

The algorithm to schedule heterogeneous jobs attempts to add jobs to an existing schedule one at a time in a non-increasing order of weights. The new job x cannot degrade performance of any job y in the existing schedule for one of the following reasons. If both jobs x and y are time-constrained, this is consistent with the optimal algorithm and theorem in Section 3.1. If both jobs are non-time-constrained, this conforms to the static priority scheduling algorithm, which is optimal for non-time-constrained jobs. If one job is time-constrained and the other is not, they are in different sets. By the definition of optimality presented in Section 1, if job 1 $\in A_j$ and job 2 $\in A_{j+k} : k > 0$, then job 1 has a greater weight, and the performance of the more important job 1 should be optimized, at the expense of job 2 where necessary. Thus, when job x is added to the schedule, time-constrained jobs in the existing schedule can only be delayed if none of them will miss their deadlines, and non-time-constrained jobs cannot be delayed at all.

The algorithm for adding a time-constrained job x is the one presented in Section 3 with a minor extension. In the algorithm in Section 3, if an attempt is made to schedule a job at a time in which another job i is currently scheduled, its deadline d_i is examined to determine whether job i can be delayed to a later slot. If job i is non-time-constrained, d_i is not defined. Since it is never appropriate to delay a non-time-constrained job in order to reduce the queueing delay of a job with a smaller weight, we let the previously undefined d_i for a non-time-constrained job i equal its ready time $r_i + 1$. (d_i should not be interpreted as a deadline for a non-time-constrained job; such jobs have no deadlines.)

Now we consider the case in which the job x to be added to the existing schedule is a non-time-constrained job. To minimize C_x, job x should be scheduled at time r_x or as soon as possible thereafter. One way to do this would be to repeatedly use the algorithm for time-constrained jobs, as follows. Set $d_x = r_x + 1$ and use the algorithm in Section 3. This will either cause job x to be scheduled at time r_x, or to be discarded. In the latter case, increment both r_x and d_x, and invoke the same algorithm again. Continue this procedure at subsequent times until job x can be scheduled. Although this would work, it could require invoking the O(N^2) algorithm to schedule a new time-constrained job as many as N times, yielding a complexity of O(N^3). A more efficient algorithm can be employed based on the following observation. As discussed in Section 3.1, the algorithm for time-constrained jobs scans time slots beginning at time r_x until a time slot $s \geq r_x$ is found which is either idle in the current schedule or for which it can be determined that none of the jobs currently scheduled at the times $\in [r_x, r_x + 1, \ldots, s]$ can be delayed without incurring some performance penalty. In the former case, a new schedule including job x has been found. In the latter case, when we are trying to schedule a time-constrained job, this means that the job should be dropped. For the non-time-constrained job in question, it means that it is not necessary to try to schedule job x at times $\in [r_x + 1, r_x + 2, \ldots, s]$. Thus, an efficient algorithm could resume its

attempts to schedule job x at time $s+1$. The resulting algorithm is shown below. Similar to the algorithm in Section 3, when inserting a new job into an existing schedule, this algorithm scans successive times until an idle time slot is found and then the new schedule can be determined. Since this requires scanning at most N times for each of the N jobs to be scheduled, the complexity is $O(N^2)$. (Again, complexity in practice is $O(BN)$ rather than the theoretical $O(N^2)$, where B is the average length of a busy period and is often independent of N.)

```
Let t = r_x                                   [t] is the job scheduled at time t
Repeat
      Clear stack
      Push x onto stack
      Let end = t+1
      While (t < end) and (slot t is not idle) do
           If (d_[t] > end) then
                Let end = d_[t]
                Push t onto stack
                Push [t] onto stack
           Let t = t + 1
Until (slot t is idle)
Pop stack into j
Schedule job j at time t
While (stack is not empty) do
      Pop stack into t
      Pop stack into j
      Schedule job j at time t
```

As an example of how one might employ this algorithm, consider an integrated services network in which voice packets arrive according to a Poisson process, generating a load of .5. Voice packets can tolerate a queueing delay of $15 - 5E$ before they are considered lost, conditioned on the fact that $15 - 5E > 0$, where E is an exponentially distributed random variable with mean 1. The scheduling algorithm must minimize $\sum w_i U_i/N$ for voice, and within that constraint, reduce the mean queueing delay $\sum_{i=1}^{N}(C_i - p_i - r_i)/N$ of data packets to the extent possible. Figure 3 shows mean queueing delay of data packets as a function of load from data packets with static priority (SP), earliest deadline first (EDF), and optimal (OPT).

Figure at end of paper.

Figure 3: Mean queueing delay $\sum_{i=1}^{N}(C_i - p_i - r_i)/N$ of non-time-constrained jobs versus load from non-time-constrained jobs, with a load from time-constrained jobs of .5.

5 Scheduling Jobs with Equal Ready Times

This section addresses the special case where $r_i = 0$ for all i. Of course, it would be possible to use the $O(N^2)$ algorithm for this problem, but as will be shown below, complexity can be reduced to $O(N \log N)$ as long as there is only one set of time-constrained jobs, i.e. the weight of any non-time-constrained job is either greater than or less than the weights of all time-constrained jobs. (A minor variation of this algorithm is also applicable when r_i can vary from job to job, but all time-constrained jobs have equal deadlines.) Since there is only one set of time-constrained jobs, there are at most three sets. Thus, there are at most three stages of the algorithm. In Stage 1, non-time-constrained jobs with large weights are scheduled. In Stage 2, the time-constrained jobs are

scheduled. Finally, in Stage 3, the remaining non-time-constrained jobs are scheduled. In Stage 1, the N_1 jobs in Set A_1 are scheduled in the first N_1 time slots in non-increasing order of weight, which is an $O(N_1 \log N_1)$ procedure.

In Stage 2, time slots are scheduled in decreasing order, beginning with the latest deadline of all of the time-constrained jobs, and working back to the earliest free time N_1. The job scheduled at time t is the one with the greatest weight of those that have not already been scheduled, and that have a deadline $\geq t+1$. The exact algorithm is shown below. It is based on heaps [21], a standard data structure in which items can be inserted in any order, and the item with the greatest value can always be retrieved. In Heap 1, it is the job with the latest deadline that can be retrieved, whereas in Heap 2, it is the job with the greatest weight. (Heap 1 is simply used to presort jobs by deadline.) The complexity of adding an item to a heap (and later removing it) is $O(\log N)$, where N is the number of items in the heap. Since each of N_2 jobs in Set A_2 will be inserted into each heap exactly once, the complexity is $O(N_2 \log N_2)$.

```
For each job i                                  H(i) is the job at the top of Heap i
      Put job i in Heap 1
Let t = d_H(1)-1
While (t ≥ N_1) do
      While (Heap 1 is not empty) and (d_H(1) = t) do
            Remove job H(1) from Heap 1 and put it in Heap 2
      If (Heap 2 is not empty) then
            Remove job H(2) from Heap 2 and schedule it at time t
      Let t = t − 1
```

We demonstrate that this algorithm is optimal for time-constrained jobs by induction. Let [t] be the job scheduled at time t using the algorithm above. We will assume that $Z_{t+1} = \sum_{i=t+1}^{\infty} w_{[i]}$, the weights of the jobs scheduled after time t, is the maximum possible, and show that $Z_t = w_{[t]} + Z_{t+1}$ is also the maximum possible. Since Z_{t+1} cannot be improved upon, the only way to increase Z_t is to replace job $[t]$ with a job of greater weight. If there are any jobs of greater weight that have deadlines $\geq t+1$, these jobs are already scheduled at a time $> t$. It would be possible to move one of these jobs with a greater weight to time t, and then replace it with an unscheduled job. However, there are no unscheduled jobs with a weight greater than that of $[t]$, so this cannot increase Z_t. Therefore Z_t is the maximum possible value for all t, and $Z_0 = \sum w_i U_i$.

Finally, in Stage 3, the less important non-time-constrained jobs are scheduled in non-increasing order of weight, in the time slots that are still idle. Time-constrained jobs were already scheduled as late as possible, so they cannot be delayed further without missing their deadlines. Sorting by weight is an $O(N_3 \log N_3)$ operation, and scanning for idle time slots is $O(N)$. Since no operation in any of the stages has a complexity greater than $O(N \log N)$, that is the complexity of the entire algorithm.

6 Summary

This paper addresses the problem of scheduling jobs with diverse performance objectives, i.e. jobs have different weights, and in some cases, different performance measures or criteria. We call such problems *heterogeneous-criteria scheduling* problems. Jobs are classified as either time-constrained,

for which $\sum w_i U_i$ should be minimized, or non-time-constrained, for which $\sum w_i C_i$ should be minimized. All jobs are assumed to be of unit length with integer ready times and deadlines. We have presented an $O(N^2)$ algorithm to schedule both time-constrained and non-time-constrained jobs of unequal weight, i.e. $1/r_i$, p_i=$1/\sum w_i U_i \& \sum w_i C_i$, where we define the objective $\gamma_1 \& \gamma_2$ in heterogeneous criteria scheduling as minimizing γ_1 for some jobs and γ_2 for others, and N is the number of jobs. We have also shown how to determine the number of jobs lost at any given weight without finding an actual schedule. In some useful cases, this can be done analytically. If not, it can be done with an $O(WN\log N)$ procedure, where W is the number of possible weights. Finally, in cases where $r_i = 0 : \forall i$, and there is only one set of time-constrained jobs, i.e. $1/p_i$=$1/\sum w_i U_i \& \sum w_i C_i$ with the one constraint on weights, we have presented an $O(N\log N)$ scheduling algorithm.

All of the algorithms in this paper have been proposed for scheduling on a single machine. However, comparable problems on M identical parallel machines can also be solved with these algorithms. Simply let $r_i = M*$(job i's ready time) and $d_i = M*$(job i's deadline), and then use the single-machine algorithms. Being scheduled at time t in the single-machine problem is then equivalent to being scheduled on machine (t mod M) at time (t div M), where machines are numbered from 0 to $M-1$. Thus, P/r_i, p_i=$1/\sum w_i U_i \& \sum w_i C_i$ is $O(N^2)$, and P/p_i=$1/\sum w_i U_i \& \sum w_i C_i$ with only one set of time-constrained jobs is $O(N\log N)$.

Although these algorithms could be applied to a wide variety of problems, the motivation for finding them came from research into both integrated-services packet-switched networks such as ATM networks, and soft real-time systems. Efficient optimal scheduling will serve as valuable tools for researchers in these contexts.

References

[1] R. L. Graham, E. L. Lawler, T. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Ann. Discrete Math.*, 5, 287-326 (1979).

[2] C. K. Chen and R. L. Bulfin, "Scheduling Unit Processing Time Jobs on a Single Machine with Multiple Criteria," *Computers Ops. Res.*, 17, 1-8 (1990).

[3] P. De, J. Ghosh, and C. E. Wells, "Some Clarifications on the Bicriteria Scheduling of Unit Execution Time Jobs on a Single Machine," *Computers Ops. Res.*, 18, 717-20 (1991).

[4] R. L. Carraway, R. J. Chambers, T. L. Morin, and H. Muskowitz, "Single Machine Sequencing with Nonlinear Multicriteria Cost Functions: An Application of Generalized Dynamic Programming," *Computers Ops. Res.*, 19, 69-77 (1992).

[5] B. Simons, "On Scheduling With Release Times and Deadlines," in *Deterministic and Stochastic Scheduling*, M. A. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan, editors. Dordrecht, The Netherlands: Reidel, 75-88 (1982).

[6] R. M. Karp, "Reducibility among Combinatorial Problems." *Complexity of Computer Communications*, R. E. Miller and J. W. Thatcher editors, Plenum Press, New York, 85-103 (1972).

[7] J. M. Peha and F. A. Tobagi, "Evaluating Scheduling Algorithms For Traffic With Heterogeneous Performance Objectives," *Proc. IEEE Globecom-90,* Dec. 1990, pp. 21-27.

[8] J. M. Peha, "Analysis of Scheduling Algorithms for Integrated-Services Networks using a Semi-Fluid-Flow Model," *Proc. IEEE Globecom-92*, Dec. 1992, pp. 330-334.

[9] J. M. Peha and F. A. Tobagi, "A Cost-Based Scheduling Algorithm To Support Integrated Services," *Proc. IEEE Infocom-91,* 741-753 (1991).

[10] J. M. Peha and F. A. Tobagi, "Cost-Based Scheduling and Dropping Algorithms To Support Integrated Services," to appear in *IEEE Trans. Communications.*

[11] L. P. Clare and A. R. K. Sastry, "Value-Based Multiplexing of Time-Critical Traffic," *Proc. IEEE Milcom-89*, 395-401 (1989).

[12] J. M. Peha, "The Priority Token Bank: Integrated Scheduling and Admission Control for an Integrated-Services Network," *Proc. IEEE Intl. Conf. Communications (ICC-93)*, Geneva, Switzerland, 345-51 (1993).

[13] R. Chipalkatti, J. F. Kurose, and D. Towsley, "Scheduling Policies for Real-Time and Non-Real-Time Traffic in a Statistical Multiplexer," *Proc. IEEE Infocom-89,* pp. 774-83 (1989).

[14] J. Hyman, A. A. Lazar, G. Pacifici, "MARS: The Magnet II Real-Time Scheduling Algorithm," *Proc. ACM Sigcomm-91,* 285-93 (1991).

[15] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model For Real-Time Operating Systems," *Proc. IEEE Real-Time Systems Symp.,* 112-122 (1985).

[16] H. Tokuda, J. W. Wendorf, and H. Y. Wang, "Implementation of a Time-Driven Scheduler for Real-Time Operating Systems," *Proc. IEEE Real-Time Systems Symp.,* 271-280 (1987).

[17] S. R. Biyabani, J. A. Stankovic, and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," *Proc. IEEE Real-Time Systems Symp.,* 142-151 (1988).

[18] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Holt, Rinehart, and Winston, New York (1976).

[19] F. Glover, "Maximum Matching in a Convex Bipartite Graph," *Nav. Res. Logist. Q.*, 14, pp. 313-316 (1967).

[20] S. S. Panwar, D. Towsley, and J. K. Wolf, "Optimal Scheduling Policies for a Class of Queues with Customer Deadlines to the Beginning of Service," *J. ACM*, 35, 832-844 (1988).

[21] D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching,* Addison-Wesley, Reading, MA, (1975).

[22] L. Dittman and S.B. Jacobsen, "Statistical Multiplexing of Identical Bursty Sources in an ATM Network," *Proc. IEEE Globecom-88,* 1293-1296 (1988).

Figure 1: Weighted loss rate, $\sum_{i=1}^{N} w_i U_i / N$, versus load.

Figure 2: Maximum load where requirements are met versus fraction of Class 1 packets, p, with traffic from 20 independent bursty sources.

Figure 3: Mean queueing delay $\sum_{i=1}^{N}(C_i - p_i - r_i)/N$ of non-time-constrained jobs versus load from non-time-constrained jobs, with a load from time-constrained jobs of .5.