Maximizing Job Benefits On-line [*]

Baruch Awerbuch [†] Yossi Azar [‡] Oded Regev [§]

November 19, 2000

Abstract

We consider a benefit model for on-line preemptive scheduling. In this model jobs arrive at the on-line scheduler at their release time. Each job arrives with its own execution time and benefit function. The flow time of a job is the time that passes from its release to its completion. The benefit function specifies the benefit gained for any given flow time. A scheduler's goal is to maximize the total benefit gained. We present a constant competitive ratio algorithm for that model in the uniprocessor case for benefit functions that do not decrease too rapidly. We also extend the algorithm to the multiprocessor case while maintaining constant competitiveness. The multiprocessor algorithm does not use migration, i.e., preempted jobs continue their execution on the same processor on which they were originally processed.

1 Introduction

1.1 The basic problem

We are given a sequence of n jobs to be assigned to one machine. Each job j has a release time r_j and a length or execution time w_j. Each job is known to the scheduler only at its release time. The scheduler may schedule the job at any time after its release time. The system allows preemption, that is, the scheduler may stop a job and later continue running it. Note that the machine

[*]A preliminary version of this paper appears in the proceedings of APPROX'00.

[†]Johns Hopkins University, Baltimore, MD 21218, and MIT Lab. for Computer Science. E-mail: baruch@blaze.cs.jhu.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARPA/Army contract DABT63-93-C-0038, ARO contract DAAL03-86-K-0171, NSF contract 9114440-CCR, DARPA contract N00014-J-92-1799, and a special grant from IBM.

[‡]Department of Computer Science, Tel Aviv University, Tel-Aviv, 69978, Israel. E-Mail: azar@math.tau.ac.il. Research supported in part by the Israel Science Foundation and by the US-Israel Binational Science Foundation (BSF).

[§]Department of Computer Science, Tel Aviv University, Tel-Aviv, 69978, Israel. E-Mail: odedr@math.tau.ac.il can process only one job at a time. If job j is completed at time c_j then we define its flow time as $f_j = c_j - r_j$ (which is at least w_j).

In the machine scheduling problem there are two major models. The first is the cost model, where the goal is to minimize the total (weighted) flow time. The second is the benefit model, where each job has its own deadline, and the goal is to maximize the benefit of jobs that meet their deadline. Both models have their disadvantages and the performance measurement is often misleading. In the cost model, a small delay in a loaded system keeps interfering with new jobs. Every new job has to wait a short while before the system is free. The result is a very large increase in total cost. This might suggest that the benefit model is favorable. However, it still lacks an important property: in many real cases, jobs are delayed by some small constant and should therefore reduce the overall system performance, but only by some small factor. In the standard benefit model, jobs that are delayed beyond their deadline cease to contribute to the total benefit. Thus, the property we are looking for is the possibility of delaying jobs without drastically harming the overall system performance.

We present a benefit model where the benefit is a function of its flow time: the longer the processing of a job takes, the lower its benefit is. More specifically, each job j has an arbitrary monotone non-increasing non-negative benefit density function $B_j(t)$ for $t \geq w_j$, and the benefit gained is $w_j B_j(f_j)$, where f_j is its flow time. Note that the benefit density function may be different for each job. The goal of the scheduler is to schedule the jobs so as to maximize the total benefit, i.e., $\sum_j w_j B_j(f_j)$, where f_j is the flow time of job j. Note that the benefit density function of different jobs can be uncorrelated and the ratio between their values can be arbitrarily large. However, we restrict each $B_j(t)$ to satisfy

$$\frac{B_j(t)}{B_j(t+w_j)} \leq C$$

for some fixed constant C. That is, if we delay a job by its length then we lose only a constant factor in its benefit.

An on-line algorithm is measured by its competitive ratio, defined as

$$\max_I \frac{OPT(I)}{A(I)},$$

where $A(I)$ denotes the benefit gained by the on-line algorithm A on input I, and $OPT(I)$ denotes the benefit gained by the optimal schedule.

As with many other scheduling problems, the uniprocessor model presented above can be extended to a multiprocessor model where instead of just one machine, we are given m identical machines. A job can be processed by at most one machine at a time. The only definition that needs further explanation is the definition of preemption. In the multiprocessor model we usually allow the scheduler to preempt a job and later continue running it on a different machine. That operation, known as migration, can be costly in many realistic multiprocessor systems. A desirable property for a multiprocessor scheduler is that it does not use migration, i.e., once a job starts running on a machine, it continues running there up to its completion. Our multiprocessor algorithm has that property with no significant degradation in performance.

1.2 The results in this paper

The main contribution of this paper is in defining a general benefit model and providing a constant competitive algorithm for this model. We begin by describing and analyzing the uniprocessor scheduling algorithm. Later, we extend the result to the multiprocessor case. Our multiprocessor algorithm does not use migration. Nevertheless, there is no such restriction on the optimal algorithm. In other words, the competitiveness result is against a possibly migrative optimal algorithm.

1.3 Previous work

The benefit model of the real-time scheduling presented above is a well-studied one. An equivalent way of looking at deadlines is to consider benefit density functions of the following 'stair' form: the benefit density for flow times which are less than or equal to a certain value, is fixed. Beyond that point, the benefit density is zero. The point of time in which the flow time of a job passes that point is the job's deadline. Such benefit density functions do not match our requirements because of their sharp decrease.

As a result of the firm deadline, the real-time scheduling model is hard to approximate. The optimal deterministic competitive ratio for the uniprocessor case is $\Theta(\Phi)$, where Φ is the ratio between the maximum and minimum benefit densities [3, 4, 7]. For the special case where $\Phi = 1$, there is a 4-competitive algorithm. The optimal randomized competitive ratio for the uniprocessor case is $O(\min(\log \Phi, \log \Delta))$, where Δ is the ratio between the longest and shortest job [6].

For the multiprocessor case, Koren and Shasha [8] showed that when the number of machines is very large, a $O(\log \Phi)$ competitive algorithm is possible. That result is shown to be optimal. Their algorithm achieves that competitive ratio without using migration.

Another related problem is the problem of minimizing the total flow time. Recall that in this problem individual benefits do not exist and the goal function is minimizing the sum (or equivalently, average) of flow times over all jobs. Unlike real-time scheduling, the uniprocessor case is solvable in polynomial time using the shortest remaining processing time first rule [2]. Using this rule, also known as SRPT, the algorithm assigns the jobs with the least remaining processing time to the available machines.

Minimizing the total flow time with more than one machine becomes $NP-hard$ [5]. In [9], Leonardi and Raz analyzed the performance of the SRPT algorithm. They showed that it achieves a competitive ratio of $O(\log(\min\{\Delta, \frac{n}{m}\}))$ where Δ is the ratio between the longest and shortest processing time. They also showed that $SRPT$ is optimal with two lower bounds for on-line algorithms, $\Omega(\log \frac{n}{m})$ and $\Omega(\log \Delta)$. A fundamental property of $SRPT$ is the use of migration. In a recent paper [1], an algorithm which achieves almost the same competitive ratio is shown. This algorithm however does not use migration.

2 The algorithm

The basic idea of the algorithm is to schedule a job whose current benefit density is as high as possible. The problem with such an algorithm is that it may preempt jobs in order to gain a small improvement in the benefit density and hence delay a large number of jobs. To overcome this problem we schedule a new job only if its benefit density is significantly higher than that of the current job. In addition, we prefer partially processed jobs to non-processed jobs of similar benefit density. The algorithm combines the above ideas and is formally described below.

We begin by defining three 'storage' locations for jobs. The first is the *pool* where new jobs arrive and stay until their processing begins. Once the scheduler decides a job should begin running, the job is removed from the pool and pushed into the *stack* where its processing begins. Two different possibilities exist at the end of a job's life cycle. The first is a job that is completed and can be popped from the stack. The second is a job that after remaining too long in the stack got thrown into the *garbage collection*. The garbage collection holds jobs whose processing we prefer to defer. The actual processing can occur when the system reaches an idle state. Throwing a job in the garbage collection means we gain nothing from it and we prefer to throw it away in order to make room for other jobs.

The job at the top of the stack is the job that is currently running. The other jobs in the stack are preempted jobs. For each job j, denote by s_j the time it enters the stack. We define its breakpoint as the time $s_j + 2w_j$. If a job is still running when it reaches its breakpoint, it is thrown into the garbage collection. We also define priorities for each job in the pool and in the stack. The priority of job j at time t is denoted by $d_j(t)$. For $t \leq s_j$, it is $B_j(t + w_j - r_j)$ and for time $t > s_j$, it is $\hat{d}_j = B_j(s_j + w_j - r_j)$. In other words, the priority of a job in the pool is its benefit density if it would have run to completion starting at the current time t. Once it enters the stack its priority becomes fixed, i.e. remains the priority at time s_j.

We describe Algorithm $ALG1$ as an event-driven algorithm. The algorithm takes action at time t when a new job is released, when the currently running job is completed or when the currently running job reaches its breakpoint. If some events happen at the same time we handle the completion of jobs first.

- A new job l arrives. If $d_l(t) > 4\hat{d}_k$, where k is the job at the top of the stack or if the stack is empty, push job l into that stack and run it. Otherwise, just add job l to the pool.

- The job at the top of the stack is completed or reaches its breakpoint. Then, pop jobs from the top of the stack and insert them into the garbage collection as long as their breakpoints have been reached. Unless the stack is empty, let k be the index of the new job at the top of the stack. Continue running job k only if $d_j(t) \leq 4\hat{d}_k$ for all j in the pool. Otherwise, get the job from the pool with maximum $d_j(t)$, push it into the stack, and run it.
- Whenever the machine is idle (i.e., no jobs in the stack or in the pool) run any uncompleted job from the garbage collection until a new job arrives.

We note several facts about this algorithm:

Observation 2.1 *Every job enters the stack at some point in time. Then, by time $s_j + 2w_j$, it is either completed or reaches its breakpoint and gets thrown into the garbage collection.*

Observation 2.2 *The priority of a job is monotone non-increasing over time. Once the job enters a stack, its priority remains fixed until it is completed or thrown away. At any time the priority of each job in a stack is at least 4 times higher than the priority of the job below it.*

Observation 2.3 *Whenever the pool is not empty, the machine is not idle, that is, the stack is not empty. Moreover, the priority of jobs in the pool is always at most 4 times higher than the priority of the currently running job.*

3 The analysis

We begin by fixing an input sequence and hence the behavior of the optimal algorithm and the on-line algorithm. We denote by f_j^{OPT} the flow time of job j by the optimal algorithm. As for the on-line algorithm, we only consider the benefit of jobs which were not thrown into the garbage collection. Denote the set of these jobs by A. So, for $j \in A$, let f_j^{ON} be the flow time of job j by the on-line algorithm. By definition,

$$V^{OPT} = \sum_j w_j B_j(f_j^{OPT})$$

and

$$V^{ON} \geq \sum_{j \in A} w_j B_j(f_j^{ON}) \ .$$

We also define the pseudo-benefit of a job j by $w_j \hat{d}_j$. That is, each job donates a benefit of $w_j \hat{d}_j$ as if it runs to completion without interruption from the moment it enters the stack. Define the pseudo-benefit of the online algorithm as

$$V^{PSEUDO} = \sum_j w_j \hat{d}_j \ .$$

For $0 \leq t < w_j$, we define $B_j(t) = B_j(w_j)$. In addition, we partition the set of jobs J into two sets, J_1 and J_2. The first is the set of jobs which are still processed by the optimal scheduler at time s_j, when they enter the stack. The second is the set of jobs which have been completed by the optimal scheduler before they enter the stack.

Lemma 3.1 *For $j \in J_1$, $\sum_{j \in J_1} w_j B_j(f_j^{OPT}) \leq C \cdot V_{PSEUDO}$.*

Proof: We note the following:

$$w_j B_j(f_j^{OPT}) \leq C \cdot w_j B_j(f_j^{OPT} + w_j) \leq C \cdot w_j B_j(s_j - r_j + w_j) = C \cdot w_j \hat{d}_j$$

where the first inequality is by our assumptions on B_j and the second is by our definition of J_1. Summing over jobs in J_1, we have

$$\sum_{j \in J_1} w_j B_j(f_j^{OPT}) \leq C \sum_{j \in J_1} w_j \hat{d}_j \leq C \cdot V_{PSEUDO}.$$

■

Lemma 3.2 *For $j \in J_2$, $\sum_{j \in J_2} w_j B_j(f_j^{OPT}) \leq 4C \cdot V^{PSEUDO}$*

Proof: For each $j \in J_2$, we define its 'optimal processing time' as

$$\tau_j = \{t | job\ j\ is\ processed\ by\ OPT\ at\ time\ t\}.$$

$$\begin{aligned}
\sum_{j \in J_2} w_j B_j(f_j^{OPT}) &= \sum_{j \in J_2} \int_{t \in \tau_j} B_j(f_j^{OPT}) dt \\
&\leq \sum_{j \in J_2} \int_{t \in \tau_j} B_j(t - r_j) dt \\
&\leq C \cdot \sum_{j \in J_2} \int_{t \in \tau_j} d_j(t) dt.
\end{aligned}$$

According to the definition of J_2, during the processing of job $j \in J_2$ by the optimal algorithm, the on-line algorithm still keeps the job in its pool. By Observation 2.3 we know that the job's priority is not too high; it is at most 4 times the priority of the currently running job and, specifically, at time $t \in \tau_j$, its priority is at most 4 times the priority of the job at the top of the stack in the on-line algorithm. Denote that job by $j(t)$. So,

$$\begin{aligned} C \cdot \sum_{j \in J_2} \int_{t \in \tau_j} d_j(t) dt &\leq 4C \cdot \sum_{j \in J_2} \int_{t \in \tau_j} \hat{d}_{j(t)} dt \\ &\leq 4C \cdot \int_{t \in \cup \tau_j} \hat{d}_{j(t)} dt \\ &\leq 4C \cdot \int_t \hat{d}_{j(t)} dt \\ &\leq 4C \cdot \sum_{j \in J} w_j \hat{d}_j = 4C \cdot V^{PSEUDO}. \end{aligned}$$

■

Corollary 3.3 $V^{OPT} \leq 5CV^{PSEUDO}$.

Proof: Combining the two lemmas we get,

$$\begin{aligned} V^{OPT} &= \sum_{j \in J_1} w_j B_j(f_j^{OPT}) + \sum_{j \in J_2} w_j B_j(f_j^{OPT}) \\ &\leq C \cdot V_{PSEUDO} + 4C \cdot V^{PSEUDO} \\ &= 5CV^{PSEUDO}. \end{aligned}$$

■

Lemma 3.4 $V^{PSEUDO} \leq 2C \cdot V^{ON}$

Proof: We show a way to divide a benefit of $C \cdot V^{ON}$ between all the jobs such that the ratio between the gain allocated to each job and its pseudo-gain is at most 2.

We begin by ordering the jobs so that jobs are preempted only by jobs appearing earlier in the order. This is done by looking at the preemption graph: each node represents a job and the directed edge (j, k) indicates that job j preempts job k at some time in the on-line algorithm. This graph is acyclic since the edge (j, k) exists only if $\hat{d}_j > \hat{d}_k$. We use a topological order of this graph in our construction. Jobs can only be preempted by jobs appearing earlier in this order.

We begin by assigning a benefit of $w_j \hat{d}_j$ to any job j in A, the set of jobs not thrown into the garbage collection. At the end of the process the benefit allocated to each job, not necessarily in A, will be at least $\frac{1}{2} w_j \hat{d}_j$.

According to the order defined above, we consider one job at a time. Assume we arrive at job j. When $j \in A$, it already has a benefit of $w_j \hat{d}_j$ assigned to it. Otherwise, job j gets thrown into the garbage collection. This job enters the stack at time s_j and leaves it at time $s_j + 2w_j$. During that time the scheduler actually processes the job for less than w_j time. So, job j is preempted for more than w_j time. For any job k running while job j is preempted, we denote by $U_{k,j}$ the set of times when job j is preempted by job k. Then, we move a benefit of $|U_{k,j}| \cdot \hat{d}_j$ from k to j. Therefore, once we finish with job j, its allocated benefit is at least $w_j \hat{d}_j$.

How much benefit is allocated to each job j at the end of the process? We have seen that before moving on to the next job, the benefit allocated to job j is at least $w_j \hat{d}_j$ (whether or not $j \in A$). When job j enters the stack at time s_j it preempts several jobs; these jobs appear later in the order. Since jobs are added and removed only from the top of the stack, as long as job j is in the stack, the set of jobs preempted by it remains unchanged. Each job k of this set gets a benefit of at most $w_j \hat{d}_k$ from j. However, since all of these jobs exist together with j in the stack at time s_j, the sum of their priorities is at most $\frac{1}{2}\hat{d}_j$ (according to Observation 2.2). So, after moving all the required benefit, job j is left with at least $\frac{1}{2} w_j \hat{d}_j$, as needed.

In order to complete the proof,

$$\begin{aligned}
V^{PSEUDO} &= \sum_j w_j \hat{d}_j = 2 \sum_j \frac{1}{2} w_j \hat{d}_j \\
&\leq 2 \sum_{j \in A} w_j \hat{d}_j \\
&\leq 2C \sum_{j \in A} w_j B_j (s_j - r_j + 2w_j) \\
&\leq 2C \sum_{j \in A} w_j B_j (f_j^{ON}) \\
&\leq 2C \cdot V^{ON}.
\end{aligned}$$

■

Theorem 3.5 *Algorithm $ALG1$ is $10C^2$ competitive.*

Proof: By combining the previous lemmas, we conclude that

$$V^{ON} \geq \frac{V^{PSEUDO}}{2C} \geq \frac{V^{OPT}}{10C^2}.$$

■

4 Multiprocessor scheduling

We extend Algorithm $ALG1$ to the multiprocessor model. In this model, the algorithm holds m stacks, one for each machine, as well as m garbage collections. Jobs not completed by their deadline get thrown into the corresponding garbage collection. Their processing can continue later when the machine is idle. As before, we assume we get no benefit from these jobs. The multiprocessor Algorithm $ALG2$ is as follows:

- A new job l arrives. If there is a machine such that $d_l(t) > 4\hat{d}_k$ where k is the job at the top of its stack or its stack is empty, push job l into that stack and run it. Otherwise, just add job l to the pool.
- The job at the top of a stack is completed or reaches its breakpoint. Then, pop jobs from the top of that stack as long as their breakpoints have been reached. Unless the stack is empty, let k be the index of the new job at the top of the stack. Continue running job k only if $d_j(t) \leq 4\hat{d}_k$ for all j in the pool. Otherwise, get the job from the pool with maximum $d_j(t)$, push it into that stack, and run it.
- Whenever a machine is idle (i.e., no jobs in its stack or in the pool) run any uncompleted job from its garbage collection until a new job arrives.

We define J_1 and J_2 in exactly the same way as in the uniprocessor case.

Lemma 4.1 *For $j \in J_1$, $\sum_{j\in J_1} w_j B_j(f_j^{OPT}) \leq C \cdot V_{PSEUDO}$.*

Proof: Since the proof of Lemma 3.1 used the definition of J_1 separately for each job, it remains true in the multiprocessor case as well. ■

The following lemma extends Lemma 3.2 to the multiprocessor case:

Lemma 4.2 *For $j \in J_2$, $\sum_{j\in J_2} w_j B_j(f_j^{OPT}) \leq 4C \cdot V^{PSEUDO}$.*

Proof: For each $j \in J_2$, we define its 'optimal processing time' by machine i as

$$\tau_{j,i} = \{t | \textit{job } j \textit{ is processed by } OPT \textit{ on machine } i \textit{ at time } t\}.$$

$$\begin{aligned}
\sum_{j\in J_2} w_j B_j(f_j^{OPT}) &= \sum_{j\in J_2}\sum_{1\leq i\leq m}\int_{t\in\tau_{j,i}} B_j(f_j^{OPT})dt \\
&\leq \sum_{j\in J_2}\sum_{1\leq i\leq m}\int_{t\in\tau_{j,i}} B_j(t-r_j)dt \\
&\leq C\cdot\sum_{j\in J_2}\sum_{1\leq i\leq m}\int_{t\in\tau_{j,i}} d_j(t)dt.
\end{aligned}$$

According to the definition of J_2, during the processing of job $j \in J_2$ by the optimal algorithm, the on-line algorithm still keeps the job in its pool. By Observation 2.3 we know that the job's priority is not too high; it is at most 4 times the priority of the currently running jobs and, specifically, at time t for machine i such that $t \in \tau_{j,i}$, its priority is at most 4 times the priority of the job at the top of stack i in the on-line algorithm. Denote that job by $j(t,i)$. So,

$$\begin{aligned} C \cdot \sum_{j \in J_2} \sum_{1 \le i \le m} \int_{t \in \tau_{j,i}} d_j(t) dt &\le 4C \cdot \sum_{j \in J_2} \sum_{1 \le i \le m} \int_{t \in \tau_{j,i}} \hat{d}_{j(t,i)} dt \\ &\le 4C \cdot \sum_{1 \le i \le m} \int_{t \in \cup \tau_{j,i}} \hat{d}_{j(t,i)} dt \\ &\le 4C \cdot \sum_{1 \le i \le m} \int_t \hat{d}_{j(t,i)} dt \\ &\le 4C \cdot \sum_{j \in J} w_j \hat{d}_j = 4C \cdot V^{PSEUDO}. \end{aligned}$$

■

Lemma 4.3 $V^{PSEUDO} \le 2C \cdot V^{ON}$.

Proof: By using Lemma 3.4 separately on each machine we obtain the same result for the multiprocessor case. ■

Combining all the results together we get

Theorem 4.4 *Algorithm ALG2 for the multiprocessor case is $10C^2$ competitive.*

References

[1] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. In *ACM Symposium on Theory of Computing (STOC)*, 1999.

[2] K.R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974.

[3] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. In *IEEE Real-Time Systems Symposium*, pages 106–115, 1991.

[4] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *32nd IEEE Annual Symposium on Foundations of Computer Science*, pages 100–110, San Juan, Puerto Rico, 1991.

[5] J. Du, J. Y. T. Leung, and G. H. Young. Minimizing mean flow time with release time constraint. *Theoretical Computer Science*, 75(3):347–355, 1990.

[6] B. Kalyanasundaram and K. Pruhs. Real-time scheduling with fault-tolerance. Technical report, Computer Science Dept. University of Pittsburgh.

[7] G. Koren and D. Shasha. D^{over}: An optimal on-line scheduling algorithm for over-loaded real-time systems. *IEEE Real-time Systems Symposium*, pages 290–299, 1992.

[8] G. Koren and D. Shasha. MOCA: a multiprocessor on-line competitive algorithm for real-time system scheduling. *Theoretical Computer Science*, 128(1–2):75–97, 1994.

[9] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 110–119, El Paso, Texas, 1997.