A Unified Approach to Approximating Resource Allocation and Scheduling

Amotz Bar-Noy* Reuven Bar-Yehuda† Ari Freund‡ Joseph (Seffi) Naor§
Baruch Schieber¶

Abstract

We present a general framework for solving resource allocation and scheduling problems. Given a resource of fixed size, we present algorithms that approximate the maximum throughput or the minimum loss by a constant factor. Our approximation factors apply to many problems, among which are: (i) real-time scheduling of jobs on parallel machines; (ii) bandwidth allocation for sessions between two endpoints; (iii) general caching; (iv) dynamic storage allocation; (v) bandwidth allocation on optical line and ring topologies. For some of these problems we provide the first constant factor approximation algorithm. Our algorithms are simple and efficient. They use the local-ratio technique and can be equivalently interpreted within the primal-dual schema.

Keywords: resource allocation, scheduling, bandwidth allocation, interval scheduling, interval graphs, scheduling with release times and deadlines.

*AT&T Shannon Lab, 180 Park Ave., P.O. Box 971, Florham Park, NJ 07932. On leave from the Electrical Engineering Department, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: *amotz@research.att.com*.

†Computer Science Department, Technion, Haifa 32000, Israel. Part of this work done while the author visited DIMACS during the summer of 1999. This research was supported by the Technion V.P.R. Fund and the New York Metropolitan Research Fund. E-mail: *reuven@cs.technion.ac.il*.

‡Computer Science Department, Technion, Haifa 32000, Israel. Work supported by Ministry of Trade and Industry's MAGNET program through the CONSIST consortium. Part of the work was done at Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: *arief@cs.technion.ac.il*.

§Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974. On leave from the Computer Science Department, Technion, Haifa 32000, Israel. E-mail: *naor@research.bell-labs.com*.

¶IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598. E-mail: *sbar@watson.ibm.com*.

1 Introduction

We study the generalized problem of resource allocation and activity scheduling. In this problem we have a set of activities competing for a reusable resource. Each activity utilizes a certain amount of the resource for the duration of its execution and frees it upon completion. The problem is to select a *feasible* subset of activities for execution, that is, a set of activities such that the total amount of resource allocated simultaneously for executing activities does not exceed the amount of resource available. A typical activity may admit several alternatives for its execution, at most one of which is to be selected. Each alternative consists of a time interval during which the activity is to take place, the resource requirement for the activity should it take place in this interval, and the profit to be gained by scheduling the activity in this interval. We allow the length of the time interval, the resource demand, and the profit to vary among different time intervals pertaining to the same activity. Thus, the actual objective is to find a feasible schedule specifying which activities are selected, and when each is to be executed, so as to maximize the profit accrued. We also consider the complement objective of minimizing the profit lost due to unscheduled activities.

This scenario models a wide range of applications. Two basic problems are immediately seen to fit in this framework: bandwidth allocation for sessions in communication networks and machine scheduling of jobs. Another, less obvious, problem that fits in this framework is the general caching problem in which a fixed size cache is used to store pages of varying size and reload cost. Finally, problems such as dynamic storage allocation, in which the resource must be allocated "contiguously", can be cast in our framework by adding a contiguity constraint.

The simplest problem that can be modeled in our framework is the problem of finding a maximum weight independent set in interval graphs [14]. This problem, which can be solved precisely and efficiently, corresponds to the case where the resource is a single machine and each activity is a task that requires the entire machine for its execution in precisely one time interval. No two tasks may run concurrently, hence feasible schedules correspond to independent sets. Although this problem is polynomial time solvable, it becomes NP-hard if, either the resource requirement, or the single-time-interval-per-activity constraint, are relaxed. If activities may require arbitrary amounts of the resource, the problem is NP-hard since it contains the NP-hard knapsack problem [12] as a special case (in which all time intervals intersect); if activities are allowed to have multiple time intervals, the problem is known as interval scheduling, which is NP-hard and in fact Max-SNP-hard [18].

1.1 Previous Work and Our contribution

We provide a unified approach to treating problems formulated in our model by using a novel technique for combining time and resource constraints. Our approach yields constant-factor approximations for all the problems we consider in this paper. For some of them this is the first constant factor approximation algorithm. Our algorithms are based on the local ratio technique [7] and they are simple and efficient. We also show how to interpret them within the primal-dual schema. We remark that obtaining a primal-dual algorithm for a (natural) maximization problem was posed recently as an open problem [20]. Following is a list, subdivided into broad categories, of the problems to which we applied our unified approach. We note that the time intervals in which an activity is allowed to be scheduled can be given either as an explicit list (called *discrete* input), or as a collection of time windows (called *continuous* input).

Machine Scheduling

The resource consists of k parallel machines and the activities are jobs to be scheduled on these machines. Each job can be scheduled in one of several time intervals. The goal is to maximize the profit of the executed jobs. There are several sub-cases to this problem.

Maximum weight independent set in interval graphs. Here, $k = 1$, and each job may be scheduled in one interval only. This problem is well known to be polynomial-time solvable [14]. Our algorithm also solves it optimally.

Interval scheduling. Here, $k = 1$, and each job may be scheduled in one of a finite set of time intervals. Spieksma [18] showed a simple greedy 1/2-approximation algorithm for the unweighted case (where *all* instances of all activities have identical profit). A 1/2-approximation factor via linear programming (LP) is implicit in [5] for the weighted case (arbitrary profits). Our algorithm achieves the same approximation factor for the weighted case combinatorially.

Single machine scheduling with release times and deadlines. Again, $k = 1$. Each job i has a *release time* r_i, a *deadline* d_i, and a *length* l_i, such that $d_i \geq r_i + l_i$. The job may be scheduled in any interval $[x, x + l_i)$ such that x is in the *time window* $[r_i, d_i - l_i]$. Bar-Noy *et al.* [5] described an LP-based 1/3-approximation algorithm for the weighted case. Our algorithm achieves $(1 - \varepsilon)/2$ approximation. It even allows jobs to have multiple time windows (as does the algorithm of [5]).

Scheduling on parallel unrelated machines. Here, k is arbitrary, and the profit gained by scheduling a job depends both on the job and on the machine to which it is assigned (the input lists the profit corresponding to each job/machine pair). Bar-Noy *et al.* [5] demonstrated a greedy 1/2-approximation algorithm for the unweighted case. For the weighted case, they gave an LP-based algorithm that achieves a factor of 1/3 for discrete input and 1/4 for continuous input. Our algorithm achieves $(1 - \varepsilon)/2$ approximation for continuous input and 1/2 for discrete input.

We note that for identical machines, i.e., when each job may be scheduled on any one of the k machines, Bar-Noy *et al.* [5] presented a greedy algorithm achieving a factor of $1 - 1/(1 + 1/k)^k$ for the unweighted case, and an LP-based algorithm achieving $1 - 1/(1 + 1/2k)^k$ for the weighted case (and continuous input). The former expression increases with k from 1/2 to $1 - 1/e \approx 0.63$, and the latter increases from 1/3 to $1 - 1/\sqrt{e} \approx 0.39$.

Maximum weight k-colorable subgraph in interval graphs. This is a special case of the previous problem in which each job may be schedule in precisely one interval. It can be solved precisely in polynomial time via minimum cost flow [2]. We achieve a $\frac{1}{2}+\frac{1}{4k-2}$-approximation factor through a much simpler and faster algorithm.

All of our algorithms for the problems listed above are in fact instances of a single parameterized algorithm, where the value of the parameter changes from problem to problem. The basic algorithm runs in polynomial time, specifically $O(n \log n)$, for discrete inputs, where n is the number of instances. To handle time windows efficiently we introduce a certain profit rounding process, which degrades the approximation factor by a factor of ε. The running time of this algorithm is $O(n^2/\varepsilon)$.

The particular instance of our algorithm for interval scheduling was also developed independently by Berman and DasGupta [8]. They also employ the same rounding idea which allows the algorithm to handle the time windows present in the single machine scheduling problem. In addition to single machine scheduling, they also considered scheduling on parallel machines, both identical and unrelated. Here, their approach deviates from ours. Rather than dealing with these problems directly, they solve them via single machine scheduling (similar to [5]). The approximation factors they achieve are $1-\left(\frac{k}{k+1}\right)^k$ for identical machines and $1/2$ for unrelated machines.

Bandwidth Allocation of Sessions in Communication Networks

In modern communication networks (e.g., ATM networks), there exists some available bandwidth between two endpoints in the network. The bandwidth allocation problem is the problem of finding the most profitable set of sessions that can utilize the available bandwidth. Our framework includes this problem, and moreover, we capture the case in which a session may have either a window of time or a finite set of intervals in which it can be scheduled. Our algorithm for this problem achieves a factor of $1/5$ (or $\frac{1}{5}(1-\varepsilon)$ if time windows are allowed). Prior to our work, no constant factor approximation algorithms for this problem were known. However, independent of our work, Phillips *et al.* [17] also obtained approximation algorithms for this problem via LP rounding. For discrete input they obtain an approximation factor of $\frac{1}{3}$ and for continuous input they obtain an approximation factor of $\frac{1}{6}$. We note that the paper of Albers *et al.* [1] implies a constant factor approximation (where the constant is at least 22) via LP rounding for the special case where each session can be scheduled in precisely one time interval [15].

General Caching

The general caching problem models situations in which a cache is to be used for pages of varying sizes and of varying (cache) reload costs. Specifically, the input consists of a cache size and a sequence of requests for pages. When a page is requested it must be loaded into the cache, unless it is already present there. Each page is characterized by its size and its *reload cost*, which is incurred whenever the page is loaded into the cache. (Requests for pages in the cache incur no cost.) Since the cache size is fixed, loading one page may necessitate evicting another. The goal is thus to find a minimum cost replacement schedule. We consider the off-line version of this problem, i.e., the case where the input is given ahead of time. As was shown in [1], the general caching problem can be modeled in our framework. Our algorithm for this problem yields a 4-approximation. (Note that since this is a minimization problem the approximation ratio is at least 1.) This improves the result of Albers *et al.* [1], who were only able to achieve an $O(1)$-approximation factor (using LP rounding) by increasing the size of the cache by $O(1)$ times the largest page size. If the cache size is not increased, they only achieve a logarithmic factor approximation.

Contiguous Allocation

Suppose the resource must be allocated in "contiguous" blocks that should remain the same until the resource is freed. For example, in the dynamic storage allocation objects are to be stored in computer memory for specified durations. Each object must be allocated a block of contiguous memory, and once the object is stored the location of its storage block may not be altered. The traditional goal in contiguous allocation has been to store all given objects in minimum size memory [13, 16]. The throughput version of this problem is, given a fixed size memory, to maximize the profit of the objects successfully stored. To the best of our knowledge there is no prior art for the throughput version of the dynamic storage allocation problem. We obtain a 1/35 factor approximation for this problem. Moreover, we solve the dynamic storage allocation problem in the general case where objects can be scheduled in more than one time interval. Informally, we use our techniques for the non-contiguous version of the problem so as to obtain a solution for the contiguous version, while losing only a constant factor. Another example of contiguous allocation is strip packing [19] where the goal is to pack rectangles into a strip. Our constant-factor bound applies to this problem as well.

The Line and Ring Topology Networks

Consider the bandwidth allocation problem where each session can be scheduled in precisely one time interval, and the start and end times of every such interval are restricted to be integral time instants. This subproblem can be cast in different terms. Consider a set of processors connected along a line where each processor is identified by its index along this line. By letting the processor identities play the role of the integral time instants in the original problem, we get the problem of bandwidth allocation for connection requests between processors on a line where each such request is for unlimited time. Clearly, all of our results regarding the time interpretation hold for the processors interpretation as well. They hold for both throughput maximization and loss minimization. An interesting application of our general framework concerns the case where processors are connected along a ring rather than a line. The ring topology is considered a viable network topology in the optical network setting and is well studied in the context of bandwidth allocation. In the ring topology a session between two processors may choose between two routes: going clockwise and going counter clockwise. In our framework, this is equivalent to two instances per activity. Most of the prior work in optical networks considered the identical width case in which each session requires one wavelength. Our solutions address the general case of more than one wavelength per session. We show how to adapt any approximation result for the throughput maximization problem on the line into one for the same problem on the ring. Given that the approximation factor for the line is $1/\rho$, the resulting factor for the ring is $1/(\rho + 1 + \varepsilon)$, where ε may be chosen arbitrarily small (the running time of the algorithm depends polynomially on $1/\varepsilon$).

2 Preliminaries

In our general framework the input consists of a set of *activities*, each requiring the utilization of a given, limited, *resource*. The amount of resource available is constant over time; we normalize it to unit size for convenience. (We remark that some of our problems can be generalized to the case where the amount of available resource may change over time.) The activities are specified as a collection of sets $\mathcal{A}_1, \ldots, \mathcal{A}_m$. Each set represents a single activity: it consists of all possible *instances* of that activity. An instance $I \in \mathcal{A}_i$ is defined by the following parameters.

1. A half-open time interval $[s(I), e(I))$ during which the activity will be executed. We call $s(I)$ and $e(I)$ the *start-time* and *end-time* of the instance.

2. The amount of resource required for the activity. We refer to this amount as the *width* of the instance and denote it $w(I)$. (Our terminology is inspired by bandwidth allocation problems.) Naturally, $0 < w(I) \leq 1$.

3. The *profit* $p(I) \geq 0$ gained by scheduling this instance of the activity.

Different instances of the same activity may have different parameters of duration, width, or profit. A *schedule* is a collection of instances. It is *feasible* if (1) it contains at most one instance of every activity, and (2) for all time instants t, the total width of the instances in the schedule whose time interval contains t does not exceed 1. In the **throughput maximization** problem we are asked to find a feasible schedule that maximizes the total profit accrued by instances in the schedule. In the **loss minimization** problem we seek a feasible schedule of minimum penalty, where the penalty of a schedule is defined as the total profit of activities *not* in the schedule. (We restrict each activity to consist of a single instance and define the profit of an activity to be the profit of its instance.) For a given profit function p, we use the term *p-profit* (or *p-penalty* in the loss minimization context) to refer to the profit with respect to p of a single instance or a set of instances.

2.1 The Local Ratio Technique

Our algorithms are based on the local ratio technique, first developed by Bar-Yehuda and Even [7], later extended by Bafna, Berman, and Fujito [3], and recently extended again by Bar-Yehuda [6]. These papers all treated minimization covering problems.

Let $\mathbf{p} \in \mathbb{R}^n$ be a profit (or penalty) vector, and let F be a set of feasibility constraints on vectors $\mathbf{x} \in \mathbb{R}^n$. A vector $\mathbf{x} \in \mathbb{R}^n$ is a *feasible solution* to a given problem $(F, \mathbf{p})$ if it satisfies all of the constraints in F. The *value* of a feasible solution $\mathbf{x}$ is the inner product $\mathbf{p} \cdot \mathbf{x}$. A feasible solution is *optimal* for a maximization (or minimization) problem if its value is maximal (or minimal) among all feasible solutions. A feasible solution $\mathbf{x}$ is an *r-approximate* solution, or simply an *r-approximation*, if $\mathbf{p} \cdot \mathbf{x} \geq$ (or $\leq$) $r \cdot \mathbf{p} \cdot \mathbf{x}^*$, where $\mathbf{x}^*$ is an optimal solution. An algorithm is said to have a *performance guarantee* of r if it always computes r-approximate solutions.

In this paper, we further extend the local ratio technique to cover problems of the form described above; that is, given a profit (or penalty) vector $\mathbf{p} \in \mathbb{R}^n$, find a solution vector $\mathbf{x}$ that maximizes (or minimizes) the inner product $\mathbf{p} \cdot \mathbf{x}$, subject to a given set F of feasibility constraints on $\mathbf{x}$.

Theorem 2.1 (Local Ratio) *Let F be a set of constraints and let $\mathbf{p}$, $\mathbf{p_1}$, and $\mathbf{p_2}$ be profit (or penalty) vectors such that $\mathbf{p} = \mathbf{p_1} + \mathbf{p_2}$. Then, if $\mathbf{x}$ is an r-approximate solution with respect to $(F, \mathbf{p_1})$ and with respect to $(F, \mathbf{p_2})$, then $\mathbf{x}$ is an r-approximate solution with respect to $(F, \mathbf{p})$.*

Proof: Let $\mathbf{x}^*$, $\mathbf{x}_1^*$, $\mathbf{x}_2^*$ be optimal solutions for $(F, \mathbf{p})$, $(F, \mathbf{p_1})$, and $(F, \mathbf{p_2})$ respectively. Then $\mathbf{p} \cdot \mathbf{x} = \mathbf{p_1} \cdot \mathbf{x} + \mathbf{p_2} \cdot \mathbf{x} \geq$ (or $\leq$) $r \cdot \mathbf{p_1} \cdot \mathbf{x}_1^* + r \cdot \mathbf{p_2} \cdot \mathbf{x}_2^* \geq$ (or $\leq$) $r \cdot (\mathbf{p_1} \cdot \mathbf{x}^* + \mathbf{p_2} \cdot \mathbf{x}^*) = r \cdot \mathbf{p} \cdot \mathbf{x}^*$ $\Box$

The Local Ratio Theorem applies to all problems in the above formulation. Note that F can include arbitrary feasibility constraints and not just linear, or linear integer, constraints. Nevertheless, all successful applications of the local ratio technique to date involve problems in which the constraints are either linear or linear integer, and this is also the case for the problems treated herein.

3 Throughput Maximization

In the throughput maximization problem we wish to find a feasible schedule of the activities that maximizes the total profit accrued. More formally, the goal is to find an optimal solution to the following integer programming problem on the boolean variables $\{x_I \mid I \in \mathcal{A}_i, 1 \leq i \leq n\}$.

$$\text{Maximize} \quad \sum_I p(I) \cdot x_I$$

$$\begin{array}{lrcl}
\text{subject to:} & & & \\
\text{For each time instant } t\text{:} & \displaystyle\sum_{I \,\mid\, s(I) \leq t < e(I)} w(I) \cdot x_I & \leq & 1. \\
\text{For each activity } \mathcal{A}_i\text{:} & \displaystyle\sum_{I \in \mathcal{A}_i} x_I & \leq & 1. \\
\text{For each instance } I\text{:} & x_I & \in & \{0, 1\}.
\end{array}$$

3.1 The Unified Algorithm

We present a generic scheme based on the local ratio technique to approximate the throughput maximization problem. We actually generalize the problem a bit and allow negative profits. Our scheme proceeds as follows.

1. Delete all instances with non-positive profit.
2. If no instances remain, return the empty schedule. Otherwise, proceed to the next step.
3. Select an instance $\widetilde{I}$ and decompose p by $p = p_1 + p_2$. The exact choice of $\widetilde{I}$ and the decomposition of p depend on the problem at hand and will be discussed shortly. (In the decomposition $p = p_1 + p_2$, the component p_2 may be non-positive.)
4. Solve the problem recursively using p_2 as the profit function. Let $\mathcal{S}'$ be the schedule returned.
5. If $\mathcal{S}' \cup \{\widetilde{I}\}$ is a feasible schedule, return $\mathcal{S} = \mathcal{S}' \cup \{\widetilde{I}\}$. Otherwise, return $\mathcal{S} = \mathcal{S}'$.

We now analyze the quality of the solution produced by the algorithm. We say that a feasible schedule is *I-maximal*, if either it contains instance I, or it does not contain I but adding I to it will render it infeasible. We are going to choose $\widetilde{I}$ and decompose p in such a way that (1) $p_2(\widetilde{I}) = 0$ and (2) for a certain r, which depends on the problem being solved, *every* $\widetilde{I}$-maximal schedule will be an r-approximation with respect to p_1.

Proposition 3.1 *Let r be a constant. Suppose that the method for choosing $\widetilde{I}$ and decomposing the profit function is such that (1) $p_2(\widetilde{I}) = 0$ and (2) for all profit functions p, every $\widetilde{I}$-maximal schedule is an r-approximation with respect to p_1. Then, the schedule $\mathcal{S}$ returned by the algorithm is an r-approximation.*

Proof: Clearly, the first step in which instances of non-positive profit are deleted does not change the optimum value. Thus, it is sufficient to show that $\mathcal{S}$ is an r-approximation with respect to the remaining instances. The proof is by induction on the number of recursive calls. At the basis of the recursion, the schedule returned is optimal (and hence an r-approximation), since no instances remain. For the inductive step, assume that $\mathcal{S}'$ is an r-approximation with respect to p_2. Then, since $p_2(\widetilde{I}) = 0$ and $\mathcal{S} \subseteq \mathcal{S}' \cup \{\widetilde{I}\}$, it follows that $\mathcal{S}$ is an r-approximation with respect to p_2. Since $\mathcal{S}$ is $\widetilde{I}$-maximal, it is also an r-approximation with respect to p_1. Thus, by the Local Ratio Theorem, it is an r-approximation with respect to p. $\Box$

It remains to specify how to determine $\widetilde{I}$ and the decomposition of the profit function. The choice of $\widetilde{I}$ is done by selecting an instance with minimum end-time among all activity instances (of all activities), breaking ties arbitrarily. To define the decomposition $p = p_1 + p_2$, we define p_1. We use a parameter $\alpha > 0$, whose value we fix for each problem separately, as follows. For a given instance I, let $\mathcal{A}(I)$ be the activity to which instance I belongs and let $\mathcal{I}(I)$ be the set of instances intersecting I but belonging to activities other than $\mathcal{A}(I)$. Then,

$$p_1(I) = p(\widetilde{I}) \cdot \begin{cases} 1 & I \in \mathcal{A}(\widetilde{I}), \\ \alpha \cdot w(I) & I \in \mathcal{I}(\widetilde{I}), \\ 0 & \text{otherwise.} \end{cases}$$

The choice of α influences the performance guarantee r we obtain. By Proposition 3.1 we only need to show that every $\widetilde{I}$-maximal schedule is an r-approximation with respect to p_1. To do so, we derive an upper bound b_{opt} on the optimum p_1-profit and a lower bound $b_{\max}$ on the p_1-profit of every $\widetilde{I}$-maximal schedule, both normalized by $p(\widetilde{I})$, which is to say that the actual bounds are $p(\widetilde{I}) \cdot b_{\max}$ and $p(\widetilde{I}) \cdot b_{\text{opt}}$. The ratio $r = b_{\max}/b_{\text{opt}}$ is then a lower bound on the performance of the algorithm. We now derive formulas for b_{opt} and $b_{\max}$. These formulae are valid for the generic problem and thus for all special cases as well. However, for specific problems within the framework we are sometimes able to show better choices of b_{opt} and $b_{\max}$.

Consider an optimal schedule. By the definition of p_1, only instances in $\mathcal{A}(\widetilde{I}) \cup \mathcal{I}(\widetilde{I})$ contribute to its p_1-profit. Since all the instances belonging to $\mathcal{I}(\widetilde{I})$ intersect at some point infinitesimally close to $e(\widetilde{I})$, the total width of such instances in the optimal schedule can be at most 1, and their contribution to the p_1-profit is therefore at most $\alpha \cdot p(\widetilde{I})$. The contribution of the instances in $\mathcal{A}(\widetilde{I})$ is at most $p(\widetilde{I})$ since a feasible schedule may contain at most one instance of each activity. Thus, we have $b_{\text{opt}} \leq 1 + \alpha$.

Turning to $\widetilde{I}$-maximal schedules, observe that any such schedule either contains an instance of $\mathcal{A}(\widetilde{I})$ or else contains a set $\mathcal{X} \neq \emptyset$ of instances intersecting $\widetilde{I}$ that prevent $\widetilde{I}$ from being added to the schedule. Let $w_{\max}$ and $w_{\min}$ be upper and lower bounds, respectively, on the width of instances in the input. Then the total width of instances in $\mathcal{X}$ is at least $1 - w(\widetilde{I}) \geq 1 - w_{\max}$, for otherwise $\widetilde{I}$ can be added without violating the feasibility of the schedule, and it is also not less than $w_{\min}$ since $\mathcal{X} \neq \emptyset$. Since $\widetilde{I}$ was chosen as an instance with minimum end-time, every instance I that intersects it must satisfy $s(I) < e(\widetilde{I}) \leq e(I)$. Hence, $\mathcal{X} \subseteq \mathcal{I}(\widetilde{I})$. Thus, we have $b_{\max} \geq \min\{1, \alpha \cdot \max\{w_{\min}, 1 - w_{\max}\}\}$. In the general case, this bound is meaningless, since $w_{\min}$ and $1 - w_{\max}$ may both be arbitrarily close to 0. However, the various problems we treat impose restrictions on the allowable widths of instances, and we can obtain meaningful bounds.

Lemma 3.2 *The approximation factor of the algorithm is at least*

$$\frac{\min\{1, \alpha \cdot \max\{w_{\min}, 1 - w_{\max}\}\}}{1 + \alpha} \tag{1}$$

3.2 Applications

The throughput maximization problem generalizes several known problems. In this section we present a selection of these problems and show how to approximate them using our scheme.

In some of the applications described below, an activity $\mathcal{A}$ is not represented by explicitly listing all the instances belonging to it, but rather by a collection of *time windows*. A time window $\mathcal{T}$ is a time interval $[s(\mathcal{T}), e(\mathcal{T}))$ or $[s(\mathcal{T}), e(\mathcal{T})]$ accompanied by three parameters: length $l(\mathcal{T})$, width $w(\mathcal{T})$, and profit $p(\mathcal{T})$. These are interpreted as follows. Every $t \in \mathcal{T}$ defines an instance with time interval $[t, t + l(\mathcal{T}))$, and all of these instances share the same width $w(\mathcal{T})$ and profit $p(\mathcal{T})$. Note that a time window must be closed on the left but may be closed or open on the right.

This sort of continuous input leads of course to problems in the implementation of the algorithm. We discuss this, as well as other implementation issues, in Section 3.3. For the time being, let us just mention that the algorithm can be implemented so that it always returns an $r(1 - \varepsilon)$-approximation, where r is the performance guarantee that could be achieved had the instances been specified explicitly (i.e., discrete input), and $\varepsilon > 0$ is an arbitrarily small constant. The running time of this implementation is polynomial in $1/\varepsilon$ and the number of time windows in the input.

3.2.1 Single machine scheduling

Here we assume that the resource is a single machine and each activity instance requires the machine. This is modeled by the condition $w(I) = 1$ for all instances I.

Maximum weight independent set in interval graphs. Consider the special case where each activity $\mathcal{A}_i$ is a singleton $\{I_i\}$. This is exactly the problem of finding a maximum-weight independent set in an interval graph, where each instance I_i corresponds to an interval. This is a well known problem, which can be solved precisely and efficiently by reducing it to the problem of finding the longest path in a DAG [14]. We claim that the unified algorithm with $\alpha = 1$ yields an optimal solution. To see this, observe that since $\mathcal{A}(\tilde{I}) = \{\tilde{I}\}$, only one instance from $\mathcal{A}(\tilde{I}) \cup \mathcal{I}(\tilde{I})$ may be scheduled in any feasible solution, hence $b_{\text{opt}} = 1$. In addition, $w_{\min} = w_{\max} = 1$, so $b_{\max} = 1$. Thus the performance guarantee is $b_{\max}/b_{\text{opt}} = 1$.

Interval scheduling. In this problem, each activity consists of a set of instances that are explicitly specified (see, e.g., [9, 10, 18]). This time we use the general bound $b_{\text{opt}} = 1 + \alpha$ and set $\alpha = 1$ so as to maximize Expression 1, given that $w_{\min} = w_{\max} = 1$. By Lemma 3.2 the performance guarantee is $1/2$.

Maximum weight throughput. Here activities are specified as time windows. This is exactly the problem of maximizing the weighted throughput of jobs with release times and deadlines on a single machine considered in [5]. Since the only difference between this problem and the previous one is that that the instances are not listed explicitly, we get a performance guarantee of $(1-\varepsilon)/2$.

3.2.2 Parallel machine scheduling

Here we assume that the resource is a set of k parallel machines. There are two sub-cases to this problem: *identical* machines where each activity instance may be assigned to any of the k machines, and *unrelated* machines where each activity instance specifies a particular machine on which it may be scheduled. (In case an instance may be scheduled on several machines we simply replicate the instance once for each machine.)

Note that our general framework does not capture this problem completely, since it does not model the condition that every activity must be carried out in its entirety on a single machine. We now argue that the solution returned by our algorithm will never force us to split an activity among several machines. This is clearly the case for unrelated machines since in this sub-case each activity instance specifies a single machine. In the case of identical machines, given a schedule returned by our algorithm, we consider the activity instances in the schedule in non-decreasing order of their start-times, and assign each activity to an arbitrary available machine. The feasibility of the schedule guarantees that at least one machine will always be available.

Identical machines. We model this case by setting the width of every instance equal to $1/k$. We use $\alpha = 1$ and $b_{\text{opt}} \leq 1 + \alpha = 2$. In addition, $b_{\max} \geq 1$ since any $\widetilde{I}$-maximal schedule that contains no instance of $\mathcal{A}(\widetilde{I})$ must contain k instances from $\mathcal{I}$. Thus we get a performance guarantee of $1/2$ (or $(1-\varepsilon)/2$ if the activities are specified as time windows).

A special case of interest is maximum weight k-colorable subgraph in interval graphs. This is just the case of identical machines with each activity consisting of a single instance. It can be solved precisely in polynomial time via minimum cost flow [2]. Our approach provides a 1/2-approximation factor through a much simpler and faster algorithm. As a matter of fact, the approximation factor for this case is $\frac{1}{2-1/k}$, since each activity consists of a single instance and thus $b_{\text{opt}} \leq 1 + \frac{k-1}{k}$.

Unrelated machines. We model this problem by the condition $w(I) = 1$ for all instances I, and modifying the definition of $\mathcal{I}(\widetilde{I})$ as follows. We define $\mathcal{I}(\widetilde{I})$ as the set of instances intersecting $\widetilde{I}$ that belong to other activities and *can be scheduled only on the same machine as* $\widetilde{I}$. In addition, we use the following criterion in the schedule construction phase. An instance may be added to the schedule if the activity to which it belongs is not currently represented in the schedule and the instance does not intersect any other instance already in the schedule that belongs on the same machine. It is easy to see that the analysis for the case of single machine scheduling carries over to this problem and thus we have a performance guarantee of $1/2$ (or $(1-\varepsilon)/2$ for time windows).

3.2.3 Bandwidth Allocation

We consider a scenario in which the bandwidth of a communication channel must be allocated to sessions. Here the resource is the channel's bandwidth and the activities are sessions to be routed through the channel. The sessions may be specified in either of two ways: (1) discrete input that lists for each call a set of intervals in which it can be scheduled, together with a width requirement and a profit for each such interval, or (2) continuous input that uses time windows.

Suppose all instances are *narrow*, i.e., have width at most $1/2$. Then, we use $\alpha = 2$ so as to maximize Expression 1, given that $w_{\max} \leq 1/2$ and $w_{\min} = 0$. Hence, Lemma 3.2 provides a performance guarantee of $1/3$ (or $(1-\varepsilon)/3$ in the continuous case).

Next, suppose all instances are *wide*, i.e., have weight greater than $1/2$. Then the problem reduces to interval scheduling (or, in the case of continuous input, to maximum weight throughput on a single machine) since no pair of intersecting instances may be scheduled together. Thus, we have a performance guarantee of $1/2$ (or $(1-\varepsilon)/2$).

Finally, to solve the problem in the general case we solve it separately for the narrow instances and for the wide instances, and return the solution of greater profit. Since either the optimum for the narrow instances is at least three fifths of the optimum (for the original problem), or the optimum for the wide jobs is at least two fifths of the optimum, the schedule returned is a 1/5-approximation (or $(1-\varepsilon)/5$).

As with machine scheduling, we can extend this problem to k *unrelated channels*, possibly with different widths, such that each activity instance may only be assigned to one particular channel. As with machine scheduling, the method of solution and the approximation factors carry over from the single channel case.

3.3 Implementation Issues

Observe that the algorithm may be implemented iteratively rather than recursively. At each iteration we delete all instances with non-positive profit; find $\widetilde{I}$, $\mathcal{A}(\widetilde{I})$, and $\mathcal{I}(\widetilde{I})$; compute $p_1(I)$ and perform $p(I) \leftarrow p(I) - p_1(I)$ for all $I \in \mathcal{A}(\widetilde{I}) \cup \mathcal{I}(\widetilde{I})$; and push $\widetilde{I}$ onto a stack. We iterate until no instances remain. We then construct the schedule by popping the activity instances off the stack and adding each to the current schedule if doing so does not violate the feasibility of the schedule.

A straightforward implementation of this algorithm (when instances are given explicitly and not as time windows) runs in $\Theta(n^2)$ time, where n is the number of instances. The bottleneck is the need to update the profits of the instances in $\mathcal{A}(\widetilde{I}) \cup \mathcal{I}(\widetilde{I})$: in the worst case the ith iteration updates the profits of $n-i$ instances. The determination of $\widetilde{I}$ and $\mathcal{I}(\widetilde{I})$ also requires $\Theta(n^2)$ time, as does the construction of the schedule.

A closer look at the algorithm reveals that the sole purpose of the stack is to reverse the order in which instances are considered in the schedule construction phase. What the algorithm really does is an initial phase, in which it scans the instances in non-decreasing end-time order, deleting some of them as it goes along, and a second phase, in which it scans the surviving instances in reverse order and constructs the schedule. Rather than using a stack, we can accomplish this by sorting the instances in a list and traversing it in both directions. This idea is only one step shy of the more powerful sweep-line approach, which can reduce the worst case time complexity to $O(n \log n)$, as we now describe.

The time interval of each instance has two endpoints. We denote the instance to which endpoint t belongs by $I(t)$. We sort the $2n$ endpoints of the instances in a list $\langle t_1, t_2, \ldots, t_{2n} \rangle$ such that $i < j$ if $t_i < t_j$ or if $t_i = t_j$ and t_i is the end-time of $I(t_i)$ and t_j is the start-time of $I(t_j)$. (Otherwise, ties are broken arbitrarily.) We then consider the endpoints in order, reducing profits as we go along and deleting an instance (i.e., deleting its two endpoints from the list) whenever its profit is found to be non-positive. Note that profits never increase, so there is no need to delete the instance the moment its profit drops to zero or less. Instead, we wait with the deletion until the end-time of the instance is encountered.

The pseudo-code below implements the sweep-line idea (though it does not achieve the promised $O(n \log n)$ time complexity). For each instance I it uses a variable π_I in which it holds the current profit of I. The value of π_I is maintained correctly only as long as I is *alive*, that is, as long as I

would still be part of the input in the recursive algorithm. To avoid confusion we stress that $p(I)$ denotes the original profit of instance I given in the input, as opposed to π_I which is the "current" profit.

1. Do $\pi_I \leftarrow p(I)$ for all I, as well as other initializations.
2. Sort the endpoints as described above.
3. For $i \leftarrow 1, 2, \ldots, 2n$:
4. $\quad$ $I \leftarrow I(t_i)$.
5. $\quad$ If t_i is the start-time of I : do nothing.
6. $\quad$ Else (t_i is the end-time of I):
7. $\quad\quad$ $P \leftarrow \pi_I$.
8. $\quad\quad$ If $P \leq 0$: delete I.
9. $\quad\quad$ Else:
10. $\quad\quad\quad$ For all $J \in \mathcal{A}(I)$: $\pi_J \leftarrow \pi_J - P$.
11. $\quad\quad\quad$ For all $J \in \mathcal{I}(I)$: $\pi_J \leftarrow \pi_J - \alpha \cdot w(J) \cdot P$.

The updating of profits still takes $\Theta(n^2)$ time in the worst case. We can reduce this to linear time by representing the profit function implicitly instead of using the variables π_I. (The initial sorting still runs in $O(n \log n)$ time.) With each activity $\mathcal{A}$ we associate a variable $\Delta_{\mathcal{A}}$ and with each instance I we associate a variable Δ_I. In addition, we maintain a variable Δ_P. All of these variables are initially set to 0. We represent the profits as follows. The current profit of a live instance $I \in \mathcal{A}$ is $p(I) - \Delta_{\mathcal{A}}$ if neither endpoint of I has been encountered, and it is $p(I) - \Delta_{\mathcal{A}} - \alpha \cdot w(I)(\Delta_P - \Delta_{\mathcal{A}}) + \Delta_I$ if its start-time has already been seen but its end-time has not. We modify the algorithm as follows. We replace "do nothing" in Line 5 with $\Delta_I \leftarrow \alpha \cdot w(I)(\Delta_P - \Delta_{\mathcal{A}})$ and we replace: Line 7 by $P \leftarrow p(I) - \Delta_{\mathcal{A}} - \alpha \cdot w(I)(\Delta_P - \Delta_{\mathcal{A}}) + \Delta_I$; Line 10 by

$$\begin{aligned} \Delta_{\mathcal{A}(I)} \leftarrow \Delta_{\mathcal{A}(I)} + P &= \Delta_{\mathcal{A}(I)} + (p(I) - \Delta_{\mathcal{A}(I)} - \alpha \cdot w(I)(\Delta_P - \Delta_{\mathcal{A}(I)}) + \Delta_I) \\ &= p(I) - \alpha \cdot w(I)(\Delta_P - \Delta_{\mathcal{A}(I)}) + \Delta_I; \end{aligned}$$

and Line 11 by $\Delta_P \leftarrow \Delta_P + P$.

When the traversal of the list is complete, we traverse the surviving endpoints in reverse direction and construct the schedule $\mathcal{S}$. We maintain a variable W and the following invariant: when endpoint t is reached, W holds the total width of instances containing t which have been added to $\mathcal{S}$.

12. $\mathcal{S} \leftarrow \emptyset$.
13. $W \leftarrow 0$.
14. Traverse the list backwards. For each endpoint t :
15. $\quad$ $I \leftarrow I(t)$.
16. $\quad$ If t is the end-time of I :
17. $\quad\quad$ If $W + w(I) \leq 1$:

$\mathcal{S} \leftarrow \mathcal{S} \cup \{I\}$.
$W \leftarrow W + w(I)$.
Delete the endpoints of the instances in $\mathcal{A}(I) - \{I\}$.
Else: delete the endpoints of I.
Else (t is the start-time of I): $W \leftarrow W - w(I)$.

3.3.1 Time Windows

In this section we refer to the straightforward stack-based iterative implementation of the algorithm described above. We use $p(\cdot)$ to denote the original profits and $\pi(\cdot)$ to denote the profits as they change throughout the execution of the algorithm.

When the input is in the form of time windows, we do not modify the profit of individual instances, but rather operate on whole windows at a time. At each iteration we delete all windows whose instances have non-positive profit, and find an instance $\widetilde{I}$ with earliest end-time among the remaining instances by considering $s(\mathcal{T}) + l(\mathcal{T})$ for all remaining windows $\mathcal{T}$. We push $\widetilde{I}$ on the stack and update the profits of the instances in $\mathcal{A}(\widetilde{I}) \cup \mathcal{I}(\widetilde{I})$. Note that to update the profits of instances in $\mathcal{I}(\widetilde{I})$ and still maintain the property that the profits of all instances in the same time window are the same we might need to first split some windows, as follows. The instances in $\mathcal{I}(\widetilde{I})$ constitute a collection of windows $\mathcal{T}$ obeying $e(\mathcal{T}) \geq e(\widetilde{I})$. For each such window $\mathcal{T}$, if $e(\mathcal{T}) = e(\widetilde{I})$ we reduce $\pi(\mathcal{T})$. Otherwise, we split $\mathcal{T}$ into two windows $\mathcal{T}' = [s(\mathcal{T}), e(\widetilde{I}))$ and the remainder $\mathcal{T}'' = \mathcal{T} - \mathcal{T}'$, and reduce $\pi(\mathcal{T}')$. We refer to this implementation as the *precise* algorithm. It is easy to see that the points at which the time windows may be split are all of the form: start-time of some window plus a finite sum of lengths of instances (not necessarily of the same activity). Since any such point can be no greater than the maximum end-time of an instance, there are only finitely many such points. Thus, our algorithm always halts in finite, if super-polynomial, time.

In order to attain polynomial running time we trade accuracy for speed. For any fixed $0 < \varepsilon < 1$ we modify the algorithm as follows. Whenever $\widetilde{I}$ is chosen such that $0 < \pi(\widetilde{I}) < \varepsilon p(\widetilde{I})$ we simply delete the window containing $\widetilde{I}$ and do not alter any profits. We call such an iteration *empty*. We refer to this algorithm as the *profit truncating* algorithm.

Let us analyze the running time of the profit truncating algorithm. We denote by n the number of windows in the input and introduce the following terminology. When a window $\mathcal{T}$ is split into $\mathcal{T}'$ and $\mathcal{T}''$ we say that one new window is *created*. We consider the original windows (in the input) to have been created initially. Also, for a given iteration, we say that $\widetilde{I}$ is *selected* in that iteration. Recall that time windows are split in such a way that the profits of all instances in the same window is the same. Hence, the algorithm deletes at least one window in each iteration, so the number of iterations is bounded by the number of windows ever created. Because of the manner in which new windows are created, it is obvious that any time instant may belong to at most n different windows, and thus the number of new windows created in the course of any single iteration is at most $n - 1$. To bound the number of windows ever created we therefore bound the number of non-empty iterations, since new windows can only be created in these iterations. Consider an original time window $\mathcal{T}$ given in the input. How many times can the algorithm select an instance belonging (originally) to $\mathcal{T}$ in non-empty iterations? Each time it does so, $\pi(\widetilde{I}) \geq \varepsilon p(\widetilde{I}) = \varepsilon p(\mathcal{T})$, since the iteration is non-empty. During the iteration the profits of all instances in $\mathcal{A}(\widetilde{I})$ decrease by $\pi(\widetilde{I})$. In particular, the profit of every surviving instance originally belonging to $\mathcal{T}$ drops by no less than $\varepsilon p(\mathcal{T})$. Thus, at most $1/\varepsilon$ such iterations may occur. It follows that the total number of non-empty iterations is at most n/ε. Hence, the number of windows ever created is at most $n + n(n-1)/\varepsilon < n^2/\varepsilon$ and this bounds the total number of iterations. The algorithm can be implemented such that the amortized time complexity of an iteration is $\mathrm{O}(1)$ if it is empty and $\mathrm{O}(n)$ if it is not, resulting in an overall time complexity of $\mathrm{O}(n^2/\varepsilon)$ for the first phase. Since only non-empty iterations push instances on the stack, and precisely one instance is pushed in each such iteration, the size of the stack cannot exceed n/ε. Thus a straightforward implementation of the second phase also requires $\mathrm{O}(n^2/\varepsilon)$ time, since the size of the schedule is never greater than n. Thus, the time complexity of the entire algorithm is $\mathrm{O}(n^2/\varepsilon)$.

We now analyze how the profit truncation affects the performance guarantee. For a given input, consider the time windows that the algorithm deletes in the course of its execution. Every deleted window is derived, by zero or more splitting operations, from a window originally present in the input. Without loss of generality let us assume that the algorithm never splits any windows (for we can always partition each window into its constituent deleted windows before commencing execution). We say that window $\mathcal{T}$ is *bad* if the algorithm deletes it in an empty iteration. Let $\varepsilon_{\mathcal{T}}$ be the value of $\pi(\mathcal{T})$ when bad window $\mathcal{T}$ is selected. Consider the following two scenarios. In the first scenario we run the profit truncating algorithm on the original input, and in the second scenario we change the profit of each bad window $\mathcal{T}$ to $p(\mathcal{T}) - \varepsilon_{\mathcal{T}}$ and then run the precise algorithm. It is easy to see that the precise algorithm in the second scenario can be made to run identically as the profit truncating algorithm in the first, except that at all times the profit of every undeleted bad window $\mathcal{T}$ in the first scenario will be higher by $\varepsilon_{\mathcal{T}}$ than its profit in the second scenario. It follows that the schedule $\mathcal{S}$ obtained in the first scenario can also be obtained in the second. Let p_{opt} and p'_{opt} denote the optimum profit in the first and second scenarios, respectively, and for every feasible schedule $\mathcal{X}$, let $p(\mathcal{X})$ and $p'(\mathcal{X})$ be the profits of $\mathcal{X}$ in both scenarios, respectively. Then, by construction, $p(\mathcal{X}) \geq p'(\mathcal{X}) \geq (1-\varepsilon)p(\mathcal{X})$ for all feasible schedules $\mathcal{X}$. It follows that $p'_{\mathrm{opt}} \geq (1-\varepsilon)p_{\mathrm{opt}}$. Let r be a performance guarantee for the precise algorithm. Then $p(\mathcal{S}) \geq p'(\mathcal{S}) \geq r \cdot p'_{\mathrm{opt}} \geq r(1-\varepsilon)p_{\mathrm{opt}}$, and thus we have a performance guarantee of $r(1-\varepsilon)$ for the profit truncating algorithm.

4 Loss Minimization

In the loss minimization problem the objective is to minimize the profit lost to unscheduled activities. In this context, we find it more appropriate to refer to *penalty* incurred rather than to profit lost. Hence we associate a penalty with each activity, and our objective is to find a feasible schedule of minimum penalty, where the penalty of a schedule is the sum of penalties of the activities **not** in the schedule. Our framework cannot model this type of penalty function in general. It can only model it in the special case where each activity contains a single instance. We therefore restrict the discussion to this case.

We actually solve a generalization of the problem in which the amount of resource may vary with time. Thus, the input consists of the activity specification as well as a positive function $Width(t)$ specifying the width (i.e., amount of resource) available at every time instant t. Accordingly, we allow arbitrary positive instance widths. A schedule is feasible if for all time instants t, the total width of instances in the schedule containing t is at most $Width(t)$.

Strictly speaking, for the problem to fit in the framework of the Local Ratio Theorem, we should define feasible solutions to be complements of feasible schedules. We find it easier, though, to keep thinking of schedules as solutions and we retain this view in the following exposition.

We present a variant of the Unified Algorithm achieving a performance guarantee of 4. Let $(\mathcal{H}, p)$ denote the input, where $\mathcal{H}$ is the description of the activities excluding the penalties, and p is the penalty function. If all the instances in $\mathcal{H}$ constitute a feasible schedule, we return this schedule. Otherwise, we decompose p by $p = p_1 + p_2$ (as described in the next paragraph) such that $p_2(I) \geq 0$ for all instances I, with equality for at least one instance. We solve recursively $(\mathcal{H}', p_2)$, where $\mathcal{H}'$ is obtained from $\mathcal{H}$ by deleting all instances whose p_2-penalty is 0, and obtain a schedule $\mathcal{S}'$. We then consider the instances in $\mathcal{H} - \mathcal{H}'$ in arbitrary order and add each to the schedule iff doing so retains the feasibility of the schedule. We return the schedule $\mathcal{S}$ thus constructed.

Let us define the decomposition of p by showing how to compute p_1. For a given time instant t, let $\mathcal{I}(t)$ be the set of intervals containing t. Define $\Delta(t) = \sum_{I \in \mathcal{I}(t)} w(I) - Width(t)$. To compute p_1, find t^* maximizing $\Delta(\cdot)$ and let $\Delta^* = \Delta(t^*)$. Assuming $\Delta^* > 0$, (otherwise the schedule containing all instances is feasible) let

$$p_1(I) = \rho \cdot \begin{cases} \min\{\Delta^*, w(I)\} & I \in \mathcal{I}(t^*), \\ 0 & \text{otherwise}, \end{cases}$$

where ρ (which depends on t^*) is the unique scaler resulting in non-negative p_2-penalties for all instances and zero p_2-penalty for at least one of them. A straightforward implementation of this algorithm runs in time polynomial in the number of activities and the number of time instants at which $Width(t)$ changes value.

To analyze the algorithm, assume (by induction) that $\mathcal{S}'$ is a 4-approximation for $(\mathcal{H}', p_2)$. It follows that $\mathcal{S}$ is a 4-approximation for $(\mathcal{H}, p_2)$. Observe (by induction on the recursion) that $\mathcal{S}$ is maximal in the sense that adding any instance to it violates its feasibility. Thus, by the Local Ratio Theorem, it suffices to show that every maximal feasible schedule is a 4-approximation for $(\mathcal{H}, p_1)$.

Observation 4.1 *Both of the following evaluate to at most $\rho\Delta^*$: (1) the p_1-penalty of any single instance, and (2) the total p_1-penalty of any collection of instances whose total width is at most Δ^*. Also, any collection of instances from $\mathcal{I}(t^*)$ whose total width is at least Δ^* has a total p_1-penalty of at least $\rho\Delta^*$.*

Consider an optimal schedule. Its p_1-penalty is the sum of the p_1-penalties of the instances in $\mathcal{I}(t^*)$ that are not in the schedule. Since all of these instance intersect at t^* their combined width must be at least Δ^*. Thus, the optimal p_1-penalty is at least $\rho\Delta^*$. Now consider any maximal feasible solution $\mathcal{M}$. We claim that it is a 4-approximation because its p_1-penalty cannot exceed $4\rho\Delta^*$. To demonstrate this, consider any time instant t. Let $\overline{\mathcal{M}}(t) = \mathcal{I}(t) - \mathcal{M}(t)$ be the set of instances containing t that are not in the schedule. We say that t is *unstable* if there is an interval $I \in \overline{\mathcal{M}}(t)$ such that adding I to $\mathcal{M}$ would violate the width constraint at t. We say that t is unstable *because of* I. Note that a single time instant may be unstable because of several different instances.

Lemma 4.2 *If t is an unstable time instant, then $\sum_{I \in \overline{\mathcal{M}}(t)} p_1(I) < 2\rho\Delta^*$.*

Proof: Let J be an instance of maximum width in $\overline{\mathcal{M}}(t)$. Then, since t is unstable, it is surely unstable because of J. This implies that $\sum_{I \in \mathcal{M}(t)} w(I) > \mathit{Width}(t) - w(J)$. Recall that $\Delta(t) = \sum_{I \in \mathcal{I}(t)} w(I) - \mathit{Width}(t)$. Thus, $\sum_{I \in \overline{\mathcal{M}}(t)} w(I) = \sum_{I \in \mathcal{I}(t)} w(I) - \sum_{I \in \mathcal{M}(t)} < w(J) + \Delta(t) \leq w(J) + \Delta^*$. Hence, $\sum_{I \in \overline{\mathcal{M}}(t)} p_1(I) \leq \rho\Delta^* + \rho\Delta^* = 2\rho\Delta^*$. □

Thus, if t^* is an unstable point, then the p_1-penalty of the schedule is $\sum_{I \in \overline{\mathcal{M}}(t^*)} p_1(I) < 2\rho\Delta^*$ and we are done. Otherwise, let $t_L < t^*$ and $t_R > t^*$ be the two unstable time instants closest to t^* on both sides (it may be that only one of them exists). The maximality of the schedule implies that every interval in $\overline{\mathcal{M}}(t^*)$ is the cause of instability of at least one time instant. Thus each such instance must contain t_L or t_R (or both). It follows that $\overline{\mathcal{M}}(t^*) \subseteq \overline{\mathcal{M}}(t_L) \cup \overline{\mathcal{M}}(t_R)$. Hence, by the previous lemma, the total p_1-penalty of these instances is less than $4\rho\Delta^*$.

4.1 Application: General Caching

In the **general caching** problem, a replacement schedule is sought for a cache that must accommodate pages of varying sizes. The input consists of a fixed cache size $S > 0$, a collection of pages $\{1, 2, \ldots, m\}$, and a sequence of n requests for pages. Each page j has a *size* $0 < s(j) \leq S$ and a *cost* $c(j) \geq 0$ associated with loading it into the cache. We assume for convenience that time is discrete and that the ith request is made at time i. (These assumptions cause no loss of generality as will become evident from our solution.) We denote by $r(i)$ the page being requested at time i. A *replacement schedule* is a specification of the contents of the cache at all times. It must satisfy the following condition. For all $1 \leq i \leq n$, page $r(i)$ is present in the cache at time i and the sum of sizes of the pages in the cache at that time is not greater than the cache size S. The initial contents of the cache (at time 0) may be chosen arbitrarily. Alternatively, we may insist that the cache be empty initially. The cost of a given replacement schedule is $\sum c(r(i))$ where the sum is taken over all i such that $r(i)$ is absent from the cache at time $i - 1$. The objective is to find a minimum cost replacement schedule.

Observe that if we have a replacement schedule that evicts a certain page at some time between two consecutive requests for it, we may as well evict it immediately after the first of these requests and bring it back only for the next request. Thus, we may restrict our attention to schedules in which for every two consecutive requests for a page, either the page remains present in the cache at all times in between the first request and the next, or it is absent from the cache at all times in between. This leads naturally to a description of the problem in terms of time intervals, and hence to a straightforward reduction of the problem to our loss minimization problem (see [1]). For the sake of completeness we recount the reduction shown in [1]. Given an instance of the general caching problem we construct an instance of our loss minimization problem as follows. We define the resource width by $Width(i) = S - s(r(i))$ for $1 \leq i \leq n$, and $Width(0) = S$ (or $Width(0) = 0$ if we want the cache to be empty initially). We define the activity instances as follows. Consider the request made at time i. Let j be the time at which the previous request for $r(i)$ is made, or $j = -1$ if no such request is made. If $j + 1 \leq i - 1$, we define an activity instance with time interval $[j + 1, i - 1]$, penalty $c(r(i))$, and width $s(r(i))$. The existence of this reduction implies a 4-approximation algorithm for the general caching problem.

5 A Primal-Dual Interpretation

Local ratio algorithms can often be reformulated in the primal-dual schema. In this section we demonstrate this for our throughput maximization algorithm.

It turns out that the natural primal-dual formulation does not map exactly onto our stack-based iterative algorithm of Section 3.3 but rather to the following slightly modified version. Let $\pi(I)$ be the current profit of instance I (as opposed to $p(I)$ which denotes its original profit given in the input). The modification is in the definition of $\mathcal{I}(\tilde{I})$. Our original definition for $\mathcal{I}(\tilde{I})$ was the set of all instances intersecting $\tilde{I}$ but *belonging to other activities*. The new definition is the set of all instances intersecting $\tilde{I}$ (including $\tilde{I}$ itself). As a result of this change, the profit of each instance I belonging to both $\mathcal{A}(\tilde{I})$ and $\mathcal{I}(\tilde{I})$ will now undergo two reductions, one reduction, due to its membership in $\mathcal{A}(\tilde{I})$ should be by the amount of $\pi(\tilde{I})$, and the second, due to its membership in $\mathcal{I}(\tilde{I})$, by the amount of $\alpha\pi(\tilde{I})w(I)$. Doing so, however, will have the undesirable effect of reducing the profit of $\tilde{I}$ to the negative value $-\alpha\pi(\tilde{I})w(\tilde{I})$. We rectify this by scaling the above reductions by $1/(1 + \alpha w(\tilde{I}))$, thus ensuring that the profit of $\tilde{I}$ drops to 0 precisely. It is not difficult to verify that in all cases (except for maximum weight independent set in interval graphs) this does not change the bound ratio $b_{\max}/b_{\text{opt}}$. Thus, our performance guarantees hold for the modified algorithm as well.

Recall the integer programming formulation of the problem presented in Section 3. Let us construct the dual program of the linear relaxation of this program. We define y_i as the dual variable corresponding to the constraint on activity $\mathcal{A}_i$ and z_t as the dual variable corresponding to the constraint at time t. Let T be the set of all start-times and end-times of all instances belonging to all activities and let $T_I = \{t \in T \mid s(I) \leq t < e(I)\}$ for all instances I.

$$\text{Minimize} \quad \sum_{i=1}^{m} y_i + \sum_{t \in T} z_t$$

subject to:

For each activity instance $I \in \mathcal{A}_i$: $\quad y_i + w(I) \sum_{t \in T_I} z_t \geq p(I)$.

For all i and t: $\quad y_i, z_t \geq 0$.

In primal-dual terms, our algorithm proceeds as follows. We initially set all dual variables to zero. We say that instance $I \in \mathcal{A}_i$ is *positive* at a given moment if $y_i + w(I) \sum_{t \in T_I} z_t < p(I)$. At each iteration the algorithm selects an instance $\widetilde{I}$ with minimum end-time among all positive instances. Let $s \in T$ be maximal such that $s < e(\widetilde{I})$. Note that by the definition of $T_{\widetilde{I}}$, s is maximal in $T_{\widetilde{I}}$ as well, and furthermore, every positive instance intersecting $\widetilde{I}$ contains s. Let

$$\delta = \frac{p(\widetilde{I}) - y_i - w(\widetilde{I}) \sum_{t \in T_{\widetilde{I}}} z_t}{1 + \alpha w(\widetilde{I})}.$$

The algorithm updates:

$$z_s \leftarrow z_s + \alpha\delta; \quad y_i \leftarrow y_i + \delta,$$

and pushes $\widetilde{I}$ on the stack. It iterates until no positive instances are left, at which time the dual variables constitute a feasible dual solution. This completes the first phase. In the second phase the algorithm pops each instance in turn of the stack and adds it to the schedule if doing so does not violate the feasibility of the schedule.

Using the combinatorial properties of the solution, as we have done in our local ratio analysis of the various problems, we can charge at least δ units of profit to the profit $p(I)$ of some instance I in the final solution for each increase of $(1 + \alpha)\delta$ in the value of the dual objective function. Thus, the profit of the schedule is at least $1/(1 + \alpha)$ times the final value of the dual objective function and this is our performance guarantee.

6 Contiguous Allocation

In problems where a given resource is to be distributed over time among consumers, the resource may be either *fungible* or *static*. Fungible resources are exemplified by the bandwidth allocation application: the "identity" of the bandwidth allocated to the sessions is irrelevant and can change over time. The only constraint is that the total bandwidth allocated at every instant does not exceed the available bandwidth. Static resources are exemplified by real memory allocation in a multi-threaded programming environment, where chunks of memory are allocated and freed by the various threads. Obviously, the total amount of memory allocated at any moment may not exceed the the amount available. In addition the memory allocated to a thread must be in contiguous addresses and must also remain in the same position until freed.

The general scheduling problem we have been dealing with so far is concerned with fungible resources . In this section we deal with allocating a resource of the second type, in particular, we consider the problem of **dynamic storage allocation (DSA)**. This problem models situations in which allocation has to be contiguous (e.g., contiguous addresses in computer memory or contiguous bands of wavelengths in wireless communication) and cannot be changed until freed. The terminology used is that of storage allocation, i.e., the resource is a contiguous storage area and activities are objects that require storing for different periods of time. We use the term *width* to refer to the sizes of the storage area and the objects. The traditional goal in DSA has been to store all objects in minimum size storage. This problem is NP-Hard [12]. Kierstead [16] describes an algorithm that uses a block of storage of size at most $6W^*$, where W^* is the maximum over all times t of the sum of the width demands at time t. We use Kierstead's result to achieve an approximation ratio of 1/35 for the throughput maximization version of DSA (with profits associated with successful allocations of storage). In fact, we solve a more general problem in which a single request for storage may offer several alternative time intervals, only one of which is to be selected. The duration, storage demand, and profit may all vary among different intervals pertaining to the same request. This is just the bandwidth allocation problem with contiguity requirements thrown in.

For simplicity, we normalize the amount of available storage to 1. We partition the instances into *wide* instances, whose widths are more than 1/2, and *narrow* instances, whose widths are at most 1/2. We solve the bandwidth allocation problem separately for the wide instances and for the narrow instances: we obtain a 1/2-approximation for the wide instances via interval scheduling, and a 1/3-approximation for the narrow instances as we have done for the bandwidth allocation problem. Let S_{w} be the solution for the wide instances and S_{n} the solution for the narrow instances. Observe that S_{w} is feasible for DSA since no two instances in it intersect. We proceed to extract from S_{n} a feasible solution for DSA, which we denote by S'_{n}, and return the solution with higher profit among S_{w} and S'_{n}.

We obtain S'_{n} from S_{n} as follows. Given the instances in S_{n}, we run Kierstead's algorithm [16] and find a DSA solution that fits S_{n} in a storage block of size 6. We cover this storage block by 11 partially overlapping strips, each of width 1: $[0\,..\,1]$, $[1/2\,..\,3/2]$, $[1\,..\,2]$, $[3/2\,..\,5/2]$, $\ldots$, $[9/2\,..\,11/2]$, $[5\,..\,6]$. Since the instances are all narrow, each is covered completely by one (or two) of these strips. Each strip defines a feasible solution for our problem consisting of the set of instances covered completely by the strip. We select S'_{n} as a solution with maximum profit among these solutions.

Since the total profit of these solutions is equal to the profit of S_{n}, which is a 1/3-approximation for bandwidth allocation, the profit of S'_{n} is a 1/33-approximation for bandwidth allocation. It is also a 1/33-approximation to DSA since the optimum profit for the DSA cannot be higher than the optimum for bandwidth allocation. Since either the profit of S'_{n} is at least 33/35 of the optimum profit for the DSA, or the optimum for S_{w} is at least 2/35 of the optimum, the better solution among S_{w} and S'_{n} is a 1/35-approximation.

7 Scheduling Sessions on Line and Ring Topologies

As mentioned in the introduction, all of our results regarding the time axis interpretation hold for the processors axis interpretation as well. They hold for both throughput maximization and loss minimization. They also hold for the contiguous case. Moreover, since processors are discrete there are no issues of continuous "windows" to contend with (although it is hard to think of any "real" application involving activities with multiple instances).

In this section we show how to adapt any approximation result for the throughput maximization problem on the line into one for the same problem on the ring. If the approximation factor for the line is $1/\rho$, the resulting factor for the ring will be $1/(\rho+1+\varepsilon)$ where ε may be chosen arbitrarily small (the running time of the algorithm depends polynomially on $1/\varepsilon$).

The idea is to cut the ring at an arbitrary point and to partition the set of instances (of all the activities) into those that do not pass through the cut and those that do. We denote the former by $\mathcal{I}'$ and the latter by $\mathcal{I}''$. The instances in $\mathcal{I}'$ define an equivalent problem on the line, whereas the instances in $\mathcal{I}''$ amount to a knapsack problem. It is well known that the knapsack problem admits a FPTAS. We obtain a $1/\rho$-approximation for $\mathcal{I}'$ and a $1/(1+\varepsilon)$-approximation for $\mathcal{I}''$, and return the better solution of the two. Since either the optimum profit of the instances in $\mathcal{I}'$ is at least $\frac{\rho}{\rho+1+\varepsilon}$ of the optimum profit (for the original problem), or the optimum profit of the instances in $\mathcal{I}''$ is at least $\frac{1+\varepsilon}{\rho+1+\varepsilon}$ of the optimum, the the better solution is a $1/(\rho+1+\varepsilon)$ approximation.

Acknowledgment We thank Sanjeev Khanna for many fruitful discussions and suggestions.

References

[1] S. Albers, A. Arora, and S. Khanna, Page replacement for general caching problems, *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 31–40, 1999.

[2] E.M. Arkin and E.B. Silverberg, Scheduling jobs with fixed start and end times, *Discrete Applied Mathematics*, Vol. 18, pp. 1–8, 1987.

[3] V. Bafna, P. Berman, and T. Fujito, A 2-approximation algorithm for the undirected feedback vertex set problem, *SIAM J. on Disc. Mathematics*, Vol. 12, pp. 289–297, 1999.

[4] A. Bar-Noy, R. Canetti, S. Kutten, Y. Mansour and B. Schieber, Competitive bandwidth allocation with preemption, *SIAM J. on Computing*, Vol. 28, pp. 1806–1828, 1999.

[5] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber, Approximating the throughput of multiple machines in real-time scheduling, *Proc. 31st Annual ACM Symposium on Theory of Computing*, pp. 622–631, 1999.

[6] R. Bar-Yehuda, One for the price of two: a unified approach for approximating covering problems, to appear, *Algorithmica.* See also *Proc. of the 1st International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, Aalborg, Denmark, July 1998.

[7] R. Bar-Yehuda and S. Even, A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, Vol. 25, pp. 27–46, 1985.

[8] P. Berman and B. DasGupta, Improvements in throughput maximization for real-time Scheduling, *Proc. 32nd Annual ACM Symposium on Theory of Computing*, 2000.

[9] T. Erlebach, Improved approximation algorithms for weighted edge-disjoint paths problems and time-constrained scheduling of weighted packets on trees and meshes. Submitted for publication, 1999.

[10] T. Erlebach and K. Jansen, Maximizing the number of connections in optical tree networks. *Proc. Int. Symposium on Algorithms and Computation (ISAAC)*, LNCS 1533, pp. 179–188, 1998.

[11] J.A. Garay, I.S. Gopal, S. Kutten, Y. Mansour and M. Yung, Efficient on-line call control algorithms, *Journal of Algorithms*, Vol. 23, pp. 180–194, 1997.

[12] M. Garey and D. Johnson, Computers and intractability: a guide to the theory of NP-completeness, *W. H. Freeman*, San Francisco, 1979.

[13] J. Gergov, Algorithms for compile-time memory optimization, *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 907–908, 1999.

[14] M. Golumbic, Algorithmic graph theory and perfect graphs. *Academic Press*, New York, 1980.

[15] S. Khanna, personal communication, 1999.

[16] H.A. Kierstead, A polynomial time approximation algorithm for dynamic storage allocation, *Discrete Mathematics*, Vol. 88, pp. 231–237, 1991.

[17] C. Phillips, R.N. Uma and J. Wein, Off-line admission control for general scheduling problems, *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 879–888, 2000.

[18] F.C.R. Spieksma, On the approximability of an interval scheduling problem. *Journal of Scheduling*, Vol. 2, pp. 215–227, 1999.

[19] A. Steinberg, A Strip-packing algorithm with absolute performance bound 2, *SIAM Journal on Computing*, Vol. 26, pp. 401–409, 1997.

[20] V. Vazirani, The primal-dual schema for approximation algorithms: where does it stand, and where can it go? *Workshop on Approximation Algorithms for Hard Problems in Combinatorial Optimization*, The Fields Institute for Research in Mathematical Sciences, September 1999.