Chapter 1
Guaranteeing Fair Service to Persistent Dependent Tasks*

Amotz Bar-Noy† Alain Mayer‡ Baruch Schieber† Madhu Sudan†

Abstract

We introduce a new scheduling problem that is motivated by applications in the area of access and flow-control of high-speed and wireless networks. An instance of the problem consists of a set of *persistent tasks* that have to be scheduled repeatedly. Each task has a demand to be scheduled "as often as possible". There is no *explicit* limit on the number of tasks that can be scheduled concurrently. However, such limits gets imposed implicitly by the fact that some tasks are in conflict and cannot be scheduled simultaneously. The conflicts are presented in the form of a conflict graph. We define parameters that quantify the *fairness* and *regularity* of a given schedule. We then proceed to show lower bounds on these parameters, and present fair and efficient scheduling algorithms for the special case where the conflict graph is an interval graph. Some of the results presented here extend to the case of perfect graphs and circular-arc graphs as well.

1 Introduction

In this paper we consider a new form of a scheduling problem which is characterized by two features:

Persistence of the tasks: A task does not simply go away once it is scheduled. Instead, each task must be scheduled infinitely many times. The goal is to schedule every task as frequently as possible.

Dependence among the tasks: Some tasks conflict with each other and hence cannot be scheduled concurrently. These conflicts are given by a conflict graph. This graph imposes constraints on the sets of tasks that may be scheduled concurrently. Note that these constraints are not based simply on the cardinality of the sets, but rather on the identity of the tasks within the sets.

We consider both the problems of *allocation*, i.e., how often should a task be scheduled and *regularity*, i.e., how evenly spaced are lengths of the intervals between successive scheduling of a specific task. We present a more formal description of this problem next and discuss our primary motivation immediately afterwards. While all our definitions are presented for general conflict graphs, our applications, bounds, and algorithms are for special subclasses – namely, perfect graphs, interval graphs and circular arc-graphs[1].

Problem statement An instance of the scheduling problem consists of a conflict graph G with n vertices. The vertices of G are the tasks to be scheduled and the edges of G define pairs of tasks that cannot be scheduled concurrently. The output of the scheduling algorithm is an infinite sequence of subsets of the vertices, $I_1, I_2, \ldots$, where I_t lists the tasks that are scheduled at time t. Notice that for all t, I_t must be an independent set of G.

In the form above, it is hard to analyze the running time of the scheduling algorithm. We consider instead a finite version of the above problem and use it to analyze the running time.

Input: *A conflict graph G and a time t.*

Output: *An independent set I_t denoting the set of tasks scheduled at time unit t.*

The objective of the scheduling algorithm is to achieve a *fair* allocation and a *regular* schedule. We next give some motivation and describe the context of our work. As we will see, none of the existing measures can appropriately capture the "goodness" of a schedule in our framework. Hence we proceed to introduce measures which allow for easier presentation of our results.

*Extended summary

†IBM – Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.
Email: {amotz,sbar,madhu}@watson.ibm.com.

‡Dept. of Computer Science, Columbia University, New York, NY 10027. Email: mayer@cs.columbia.edu. Part of this work was done while the author was at the IBM T. J. Watson Research Center. Partially supported by an IBM Graduate Fellowship, NSF grant CCR-93-16209, and CISE Institutional Infrastructure Grant CDA-90-24735

[1] A graph is perfect if for all its induced subgraphs the size of the maximum clique is equal to the chromatic number (cf. [11]). A graph is an interval graph (circular-arc graph) if its vertices correspond to intervals on a line (circle), and two vertices are adjacent if the corresponding intervals intersect (cf. [20]).

1.1 Motivation

Session scheduling in high-speed local-area networks. MetaRing ([7]) is a recent high-speed local-area ring-network that allows "spatial reuse", i.e., concurrent access and transmission of user sessions, using only minimal intermediate buffering of packets. The basic operations in MetaRing can be approximated by the following: if some node has to send data to some other node a *session* is established between the source and the destination. Sessions typically last for a while and can be active only if they have exclusive use of all the links in their routes. Hence, sessions whose routes share at least one link are in conflict. These conflicts need to be regulated by breaking the data sent in a session into units of quotas that are transmitted according to some schedule. This schedule has to be *efficient* and *fair*. Efficient means that the total number of quotas transmitted (throughput) is maximized whereas fair means that the throughput of *each* session is maximized, and that the time between successive activation of a session is minimized, so that large buffers at the source nodes can be avoided. It has been recognized ([5]) that the access and flow-control in such a network should depend on locality in the conflict graph. However, no firm theoretical basis for an algorithmic framework has been proposed up to now. To express this problem as our scheduling problem we create a circular-arc graph whose vertices are the sessions, and in which vertices are adjacent if the corresponding paths associated with the sessions intersect in a link.

Time sharing in wireless networks. Most indoor designs of wireless networks are based on a *cellular* architecture with a very small cell size. (See, e.g., [13].) The cellular architecture comprises two levels – a stationary level and a mobile level. The stationary level consists of fixed *base stations* that are interconnected through a *backbone* network. The mobile level consists of mobile units that communicate with the base stations via wireless links. The geographic area within which mobile units can communicate with a particular base station is referred to as a *cell*. Neighboring cells overlap with each other, thus ensuring continuity of communications. The mobile units communicate among themselves, as well as with the fixed information networks, through the base stations and the backbone network. The continuity of communications is a crucial issue in such networks. A mobile user who crosses boundaries of cells should be able to continue its communication via the new base-station. To ensure this, base-stations periodically need to transmit their identity using the wireless communication. In some implementations the wireless links use infra-red waves. Therefore, two base-station whose cells overlap are in conflict and cannot transmit their identity simultaneously. These conflicts have to be regulated by a time-sharing scheme. This time sharing has to be *efficient* and *fair*. Efficient means that the scheme should accommodate the maximal number of base stations whereas fair means that the time between two consecutive transmissions of the same base-station should be less then the time it takes a user to cross its corresponding cell. Once again this problem can be posed as our graph-scheduling problem where the vertices of the graph are the base-stations and an edge indicates that the base stations have overlapping cells.

1.2 Relationship to past work

Scheduling problems that only consider either *persistence* of the tasks or *dependence* among the tasks (but not both) have been dealt with before.

The task of scheduling persistent tasks has been studied in the work of Baruah *et al.* [2]. They consider the problem of scheduling a set of n tasks with given (arbitrary) frequencies on m machines. (Hence, $m = 1$ yields an instance of our problem where the conflict graph is a clique.) To measure "regularity" of a schedule for their problem they introduce the notion of *P-fairness*. A schedule for this problem is P-fair (proportionate-fair) if at each time t for each task i the absolute value of the difference in the number of times i has been scheduled and $f_i t$ is strictly less than 1, where f_i is the frequency of task i. They provide an algorithm for computing a P-fair solution to their problem. Their problem fails to capture our situation due to two reasons. First, we would like to constrain the sets of tasks that can be scheduled concurrently according to the topology of the conflict graph and not according to their cardinality. Moreover, in their problem every "feasible" frequency requirement can be scheduled in a P-fair manner. For our scheduling problem we show that such a P-fair schedule cannot always be achieved. To deal with feasible frequencies that cannot be scheduled in a P-fair manner, we define weaker versions of "regularity".

The dependency property captures most of the work done based on the well-known "Dining Philosophers" paradigm, see for example [9], [18], [6], [1], [8], and [4]. In this setting, Lynch [18] was the first to explicitly consider the *response time* for each task. The goal of successive works was to make the response time of a node to depend only on its local neighborhood in the conflict graph. (See, e.g., [4].) While *response time* in terms of a node's degree is adequate for "one-shot" tasks, it does not capture our requirement that a task

should be scheduled in a regular and fair fashion over a period of time.

1.3 Notations and definitions

A *schedule* S is an infinite sequence of independent sets $I_1, I_2, \ldots, I_t, \ldots$ We use the notation $S(i,t)$ to represent the schedule: $S(i,t) = 1$ if $i \in I_t$ and 0 otherwise. Let $f_i^{(t)} = \sum_{\tau=1}^{t} S(i,\tau)/t$. Let $f_i = \liminf_{t\to\infty}\{f_i^{(t)}\}$. We refer to f_i as the *frequency* of the i-th task in schedule S.

DEFINITION 1.1. *A vector of frequencies* $\hat{f} = (f_1, \ldots, f_n)$ *is* feasible *if there exists a schedule* S *such that the frequency of the* i-th *task under schedule* S *is at least* f_i.

DEFINITION 1.2. *A schedule* S realizes *a vector of frequencies* $\hat{f}$ *if the frequency of the* i-th *task under schedule* S *is at least* f_i. *A schedule* S c-approximates *a vector of frequencies* $\hat{f}$ *if the frequency of the* i-th *task under schedule* S *is at least* f_i/c.

A measure of fairness Fairness is determined via a partial order $\prec$ that we define on the set of frequency vectors.

DEFINITION 1.3. *Given two frequency vectors* $\hat{f} = (f_1, \ldots, f_n)$ *and* $\hat{g} = (g_1, \ldots, g_n)$, $\hat{f} \prec \hat{g}$ *(*$\hat{f}$ *is less fair than* $\hat{g}$*) if there exists an index* j *and a threshold* f *such that* $f_j < f \leq g_j$ *and for all* i *such that* $g_i \leq f$, $f_i \leq g_i$.

DEFINITION 1.4. *A vector of frequencies* $\hat{f}$ *is* max-min fair *if no feasible vector* $\hat{g}$ *satisfies* $\hat{f} \prec \hat{g}$.

Less formally, in a max-min fair frequency vector one cannot increase the frequency of some task at the expense of more frequently scheduled tasks. This means that our goal is to let task i have more of the resource as long as we have to take the resource away only from tasks which are better off, i.e., they have more of the resource than task i.

Measures of regularity Here, we provide two measures by which one can evaluate a schedule for its *regularity*. We call these measures the *response time* and the *drift*.

Given a schedule S, the *response time* for task i, denoted r_i, is the largest interval of time for which the i-th task waits between successive schedulings. More precisely,

$$r_i = \max\{t_2 - t_1 | 0 \leq t_1 < t_2 \text{ s.t. } \forall_{t_1<t<t_2} S(i,t) = 0\}.$$

For any time t, the number of expected occurrences of task i can be expressed as $f_i t$. But note that if r_i is larger than $1/f_i$, it is possible that, for some period of time, a schedule allows a task to "drift away" from its expected number of occurrences. In order to capture this, we introduce a second measure for the regularity of a schedule. We denote by d_i the *drift* of a task i. It indicates how much a schedule allows task i to drift away from its expected number of scheduled units (based on its frequency):

$$d_i = \max_t\{|f_i \cdot t - \sum_{r=1}^{t} S(i,r)|\}.$$

Note that if a schedule S achieves drift $d_i < 1$ for all i, then it is *P-fair* as defined in [2].

Finally, a schedule achieves its strongest form of regularity if each task i is scheduled every $1/f_i$ time-units (except for its first appearance). Hence we say that a schedule is *rigid* if for each task i there exists a starting point s_i such that the task is scheduled on exactly the time units $s_i + j(1/f_i)$, for $j = 0, 1, \ldots$

1.4 Results

In Section 2 we motivate our definition of max-min fairness and show several of its properties. First, we provide an equivalent alternate definition of feasibility which shows that deciding feasibility of a frequency vector is computable. We prove that every graph has a *unique* max-min fair frequency vector. Then, we show that the task of even weakly-approximating the max-min fair frequencies on general graphs is NP-hard. As we mentioned above many practical applications of this problem arise from simpler networks, such as buses and rings (i.e., interval conflict graphs and circular-arc conflict graphs). For the case of perfect-graphs (and hence for interval graphs), we describe an efficient algorithm for computing max-min fair frequencies. We prove that the period T of a schedule realizing such frequencies satisfies $T = 2^{O(n)}$ and that there exist interval graphs such that $T = 2^{\Omega(n)}$.

The rest of our results deal with the problem of finding the most "regular" schedule (under the above mentioned measures) that realizes any feasible frequency vector. Section 3 shows the existence of interval graphs for which there is no P-fair schedule that realizes their max-min fair frequencies. In Section 4 we introduce an algorithm for computing a schedule that realizes any given feasible frequencies on interval graphs. The schedule computed by the algorithm achieves response-time of $\lceil 4/f_i \rceil$ and drift of $O(\sqrt{\log T} n^{\epsilon})$. A slight modification of this algorithm yields a schedule that 2-approximates the given frequencies. The advantage of this schedule is that

it achieves a bound of 1 on the drift and hence a bound of $\lceil 2/f_i \rceil$ on the response time. In Section 5 we present an algorithm for computing a schedule that 12-approximates any given feasible frequencies on interval graphs and has the advantage of being rigid. All algorithms run in polynomial time. In Section 6 we show how to transform any algorithm for computing a schedule that c-approximates any given feasible frequencies on interval graphs into an algorithm for computing a schedule that $2c$-approximates any given feasible frequencies on circular-arc graphs. (The response-time and drift of the resulting schedule are doubled as well.) Finally, in Section 7 we list a number of open problems and sketch what additional properties are required to obtain solutions for actual networks. Due to space constraints some of the proofs are either omitted or sketched in this extended summary.

2 Max-min Fair Allocation

Our definition for max-min fair allocation is based on the definition used by Jaffe [14] and Bertsekas and Gallager [3], but differs in one key ingredient – namely our notion of feasibility. We study some elementary properties of our definition in this section. In particular, we show that the definition guarantees a unique max-min fair frequency vector for every conflict graph. We also show the hardness of computing the frequency vector for general graphs. However, for the special case of perfect graphs our notion turns out to be the same as of [3].

The definition of [14] and [3] is considered the traditional way to measure throughput fairness and is also based on the partial order $\prec$ as used in our definition. The primary difference between our definition and theirs is in the definition of *feasibility*. Bertsekas and Gallager [3] use a definition, which we call clique feasible, that is defined as follows:

> A vector of frequencies $(f_1, \ldots, f_n)$ is *clique feasible* for a conflict graph G, if $\sum_{i \in C} f_i \leq 1$ for all cliques C in the graph G.

The notion of max-min fairness of Bertsekas and Gallager [3] is now exactly our notion, with feasibility replaced by clique feasibility.

The definition of [3] is useful for capturing the notion of fractional allocation of a resource such as bandwidth in a communication networks. However, in our application we need to capture a notion of integral allocation of resources and hence their definition does not suffice for our purposes. It is easy to see that every frequency vector that is feasible in our sense is clique feasible. However, the converse is not true. Consider the case where the conflict graph is the five-cycle. For this graph the vector $(1/2, 1/2, 1/2, 1/2, 1/2)$ is clique feasible, but no schedule can achieve this frequency.

2.1 An alternate definition of feasibility

Given a conflict graph G, let $\mathcal{I}$ denote the family of all independent sets in G. For $I \in \mathcal{I}$, let $\chi(I)$ denote the characteristic vector of I.

PROPOSITION 2.1. *A vector of frequencies $\hat{f}$ is feasible if and only if there exist weights $\{\alpha_I\}_{I \in \mathcal{I}}$, such that $\sum_{I \in \mathcal{I}} \alpha_I = 1$ and $\sum_{I \in \mathcal{I}} \alpha_I \chi(I) = \hat{f}$.*

The main impact of this assertion is that it shows that the space of all feasible frequencies is well behaved (i.e., it is a closed, connected, compact space). Immediately it shows that determining whether a frequency vector is feasible is a computable task (a fact that may not have been easy to see from the earlier definition). We now use this definition to see the following connection:

PROPOSITION 2.2. *Given a conflict graph G, the notions of feasibility and clique feasibility are equivalent if and only if G is perfect.*

Proof (sketch): The proof is follows directly from well-known polyhedral properties of perfect graphs. (See [12], [16].) In the notation of Knuth [16] the space of all feasible vectors is the polytope $\mathsf{STAB}(G)$ and the space of all clique-feasible vectors is the polytope $\mathsf{QSTAB}(G)$. The result follows from the theorem on page 38 in [16] which says that a graph G is perfect if and only if $\mathsf{STAB}(G) = \mathsf{QSTAB}(G)$. □

2.2 Uniqueness and computability of max-min fair frequencies

In the full paper we prove the following theorem.

THEOREM 2.3. *There exists a unique max-min fair frequency vector.*

Now, we turn to the issue of the computability of the max-min fair frequencies. While we do not know the exact complexity of computing max-min fair frequencies[2] it does seem to be a very hard task in general. In particular, we consider the problem of computing the smallest frequency assigned to any vertex by a max-min allocation and show the following:

THEOREM 2.4. *There exists an $\epsilon > 0$, such that given a conflict graph on n vertices approximating the*

[2] In particular, we do not know if deciding whether a frequency vector is feasible is in NP ∪ coNP

smallest frequency assigned to any vertex in a max-min fair allocation, to within a factor of n^ϵ, is NP-hard.

Proof (sketch): We relate the computation of max-min fair frequencies in a general graph to the computation of the fractional chromatic number of a graph. The fractional chromatic number problem (cf. [17]) is defined as follows: To each independent set I in the graph, assign a weight w_I, so as to minimize the quantity $\sum_I w_I$, subject to the constraint that for every vertex v in the graph, the quantity $\sum_{I \ni v} w_I$ is at least 1. The quantity $\sum_I w_I$ is called the *fractional chromatic number* of the graph. Observe that if the w_I's are forced to be integral, then the fractional chromatic number is the chromatic number of the graph.

The following claim shows a relationship between the fractional chromatic number and the assignment of feasible frequencies.

CLAIM 2.5. *Let $(f_1, f_2, \ldots, f_n)$ be a feasible assignment of frequencies to the vertices in a graph G. Then $1/(\min_i f_i)$ is an upper bound on the fractional chromatic number of the graph. Conversely, if k is the fractional chromatic number of a graph, then a schedule that sets the frequency of every vertex to be $1/k$ is feasible.*

The above claim, combined with the hardness of computing the fractional chromatic number [17], suffices to show the NP-hardness of deciding whether a given assignment of frequencies is feasible for a given graph. To show that the claim also implies the hardness of approximating the smallest frequency in the max-min fair frequency vector we inspect the Lund-Yannakakis construction a bit more closely. Their construction yields a graph in which every vertex participates in a clique of size k such that deciding if the (fractional) chromatic number is k or kn^ϵ is NP-hard. In the former case, the max-min fair frequency assignment is $1/k$ to every vertex. In the latter case at least some vertex will have frequency smaller that $1/(kn^\epsilon)$. Thus this implies that approximating the smallest frequency in the max-min fair frequencies to within a factor of n^ϵ is NP-hard. □

2.3 Max-min fair frequencies on perfect graphs

We now turn to perfect graphs. We show how to compute in polynomial time max-min fair frequencies for this class of graphs and give bounds on the period of a schedule realizing such frequencies. As our main focus of the subsequent sections will be interval graphs, we will give our algorithms and bounds first in terms of this subclass and then show how to generalize the results to perfect graphs.

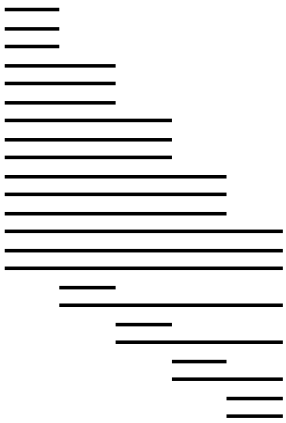

Figure 1: An interval graph for which $T = 2^{\Omega(n)}$.

We start by describing an algorithm for computing max-min fair frequencies on interval graphs. As we know that clique-feasibility equals feasibility (by Proposition 2.2), we can use an adaptation of [3]:

Algorithm 1: Let $\mathcal{C}$ be the collection of maximal cliques in the interval graph. (Notice that $\mathcal{C}$ has at most n elements and can be computed in polynomial time.) For each clique $C \in \mathcal{C}$ the algorithm maintains a *residual capacity* which is initially 1. To each vertex the algorithm associates a label *assigned/unassigned*. All vertices are initially unassigned. Dividing the residual capacity of a clique by the number of unassigned vertices in this clique yields the *relative* residual capacity. Iteratively, we consider the clique with the smallest current relative residual capacity and assign to each of the clique's unassigned vertices this capacity as its frequency. For each such vertex in the clique we mark it assigned and subtract its frequency from the residual capacity of every clique that contains it. We repeat the process till every vertex has been assigned some frequency.

It is not hard to see that Algorithm 1 correctly computes max-min fair frequencies in polynomial-time. We now use its behavior to prove a tight bound on the period of a schedule for an interval graph. The following theorem establishes this bound. (See also Figure 1.)

THEOREM 2.6. *Let $f_i = p_i/q_i$ be the frequencies in a max-min fair schedule for an interval graph G, where p_i and q_i are relatively prime. Then, the period for the schedule $T = lcm_{i=1}^{n}\{q_i\}$ satisfies $T = 2^{O(n)}$. Furthermore, there exist interval graphs for which $T = 2^{\Omega(n)}$.*

It is clear that Algorithm 1 works for all graphs where clique feasibility determines feasibility, i.e., perfect graphs. However, the algorithm does not remain computationally efficient. Still, Theorem 2.6 can be directly extended to the class of perfect graphs. We now use this fact to describe a polynomial-time algorithm for assigning max-min fair frequencies to perfect graphs.

Algorithm 2: This algorithm maintains the labelling procedure *assigned/unassigned* of Algorithm 1. At each phase, the algorithm starts with a set of assigned frequencies and tries to find the largest f such that all unassigned vertices can be assigned the frequency f. To compute f in polynomial time, the algorithm uses the fact that deciding if a given set of frequencies is feasible is reducible to the task of computing the size of the largest weighted clique in a graph with weights on vertices. The latter task is well known to be computable in polynomial-time for perfect graphs. Using this decision procedure the algorithm performs a binary search to find the largest achievable f. (The binary search does not have to be too refined due to Theorem 2.6). Having found the largest f, the algorithm finds a set of vertices which are saturated under f as follows: Let ϵ be some small number, for instance $\epsilon = 2^{-n^2}$ is sufficient. Now it raises, one at a time, the frequency of each unassigned vertex to $f+\epsilon$, while maintaining the other unassigned frequencies at f. If the so obtained set of frequencies is not feasible, then it marks the vertex as assigned and its frequency is assigned to be f. The algorithm now repeats the phase until all vertices have been assigned some frequency.

3 Non-existence of P-fair allocations

Here we show that a P-fair scheduling realizing max-min fair frequencies need not exist for every interval graph.

THEOREM 3.1. *There exist interval graphs G for which there is no P-fair schedule that realizes their max-min frequency assignment.*

In order to prove this theorem we construct such a graph G as follows. We choose a parameter k and for every permutation π of the elements $\{1,\ldots,k\}$, we define an interval graph G_π. We show a necessary condition that π must satisfy if G_π has a P-fair schedule. Lastly we show that there exists a permutation π of 12 elements which does not satisfy this condition.

Given a permutation π on k elements, G_π consists of $3k$ intervals. For $i \in \{1,\ldots,k\}$, we define the intervals $A(i) = (i-1, i]$, $B(i) = (i, k+\pi(i)+1]$ and $C(i) = (k+i+1, k+i+2]$. Observe that the max-min frequency assignment to G_π is the following: All the tasks $B(i)$ have frequency $1/k$; all the tasks $A(i)$ have frequency $(k-i+1)/k$; and all the tasks $C(i)$ have frequency i/k. (See Figure 2.)

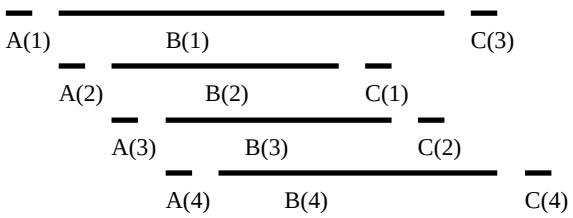

Figure 2: The graph G_π for $\pi = (3,1,2,4)$

We now observe the properties of a P-fair schedule for the tasks in G_π. (i) The time period is k. (ii) The schedule is entirely specified by the schedule for the tasks $B(i)$. (iii) This schedule is a permutation σ of k elements, where $\sigma(i)$ is the time unit for which $B(i)$ is scheduled. To see what kind of permutations σ constitute P-fair schedules of G_π we define the notion of when a permutation is fair for another permutation.

DEFINITION 3.1. *A permutation σ_1 is* fair *for a permutation σ_2 if for all i,j, $1 \le i,j \le k$, σ_1 and σ_2 satisfy the conditions* cond_{ij} *defined as follows:*

$$\frac{ij}{k} - 1 < |\{l : \sigma_2(l) \le i, \sigma_1(l) \le j\}| < \frac{ij}{k} + 1 \ .$$

CLAIM 3.2. *If a permutation σ is a P-fair schedule for G_π then σ is fair for the identity permutation and for permutation π.*

Let $\pi = (1,3,4,7,8,9,11,5,12,10,2,6)$ be a permutation on 12 elements. In the full paper we show that no permutation σ is fair to both π and the identity.

4 Realizing frequencies exactly

In this section we first show how to construct a schedule that realizes any feasible set of frequencies (and hence in particular max-min frequencies) exactly on an interval graph. We prove its correctness and demonstrate a bound of $\lceil 4/f_i \rceil$ on the response time for each interval i. We then proceed to introduce a potential function that is used to yield a bound of $O(n^{\frac{1}{2}+\epsilon})$ on the drift for every interval. We also prove that if the feasible frequencies are of the form $1/2^i$, then the drift of the schedule can be bounded by 1 and thus the waiting time can be bounded by $\lceil 2/f_i \rceil$. We use this property to give an algorithm for

computing a schedule that 2-approximates any feasible set of frequencies with high regularity.

Input to the Algorithm: A unit of time t and a conflict graph G which is an interval graph. Equivalently, a set $I = \{I_1, \ldots, I_n\}$ of intervals on the unit interval $[0, 1]$ of the x-coordinate, where $I_i = [i.s, i.e]$ for $1 \leq i \leq n$. Every interval I_i has a frequency $f_i = p_i/q_i$ with the following constraint: $\sum_{I_i \ni x} f_i \leq 1$ for all $0 \leq x \leq 1$. For simplicity, we assume from now on that these constraints on the frequencies are met with equality and that $t \leq T = LCM\{q_i\}$.

Output of the Algorithm: An independent set I_t which is the set of tasks scheduled for time t such that the scheduled S, given by $\{I_t\}_{t=1}^{T}$ realizes frequencies f_i.

The algorithm is recursive. Let s_i denote the number of times a task i has to appear in T time units, i.e., $s_i = Tp_i/q_i$. The algorithm has $\log T$ levels of recursion. (Recall that $\log T$ is $O(n)$ for max-min fair frequencies.) In the first level we decide on the occurrences of the tasks in each half of the period. That is, for each task we decide how many of its occurrences appear in the first half of the period and how many in the second half. This yields a problem of a recursive nature in the two halves. In order to find the schedule at time t, it suffices to solve the problem recursively in the half which contains t. (Note that in case T is odd one of the halves is longer than the other.) Clearly, if a task has an even number of occurrences in T it would appear the same number of times in each half in order to minimize the drift. The problem is with tasks that have an odd number of occurrences s_i. Clearly, each half should have at least $\lfloor s_i \rfloor$ of the occurrences. The additional occurrence has to be assigned to a half in such a way that both resulting sub-problems would still be feasible. This is the main difficulty of the assignment and is solved in the procedure *Sweep*.

Procedure Sweep: In this procedure we compute the assignment of the additional occurrence for all tasks that have an odd number of occurrences. The input to this procedure is a set of intervals $I_1, \ldots, I_m$ (with odd s_i's) with the restriction that each clique in the resulting interval graph is of even size. (Later, we show how to overcome this restriction.) The output is a partition of these intervals into two sets such that each clique is equally divided among the sets. This is done by a sweep along the x-coordinate of the intervals. During the sweep every interval will be assigned a variable which at the end is set to 0 or 1 (i.e., first half of the period or second half of the period). Suppose that we sweep point x. We say that an interval I_i is *active* while we sweep point x if $x \in I_i$. The assignment rules are as follows.

For each interval I_i that starts at x:

If the current number of active intervals is even:
A new variable is assigned to I_i (I_i is *unpaired*).
If the current number of active intervals is odd:
I_i is paired to the currently unpaired interval I_j and it is assigned the negation of I_j's variable. Thus no matter what value is later assigned to this variable, I_i and I_j will end up in opposite halves.

For each interval I_i that ends at x:

If the current number of active intervals is even:
Nothing is done.
If the current number of active intervals is odd:
If I_i is paired with I_j:
I_j is now being paired with the currently unpaired interval I_k. Also, I_j's variable is matched with the negation of I_k's variable. This will ensure that I_j and I_k are put in opposite halves, or equivalently, I_i and I_k are put in the same halves.
If I_i is unpaired:
Assign arbitrarily 0 or 1 to I_i's variable.

These operations ensure that whenever the number of active intervals is even, then exactly half of the intervals will be assigned 0 and half will be assigned 1; this will be proven later.

Recall that we assumed that the size of each clique is even. Let us show how to overcome this restriction. For this we need the following simple lemma. For $x \in [0, 1]$, denote by C_x the set of all the input intervals (with odd and even s_i's) that contain x; C_x will be referred to as a clique.

LEMMA 4.1. *The period T is even if and only if $|\{i \ : \ I_i \in C \wedge s_i \text{ is odd}\}|$ is even for every clique C.*

This lemma implies that if T is even then the size of each clique in the input to procedure *Sweep* is indeed even. If T is odd, then a dummy interval I_{n+1} which extends over all other intervals and which has exactly one occurrence is added to the set I before calling *Sweep*. Again, by Lemma 4.1, we are sure that in this modified set I the size of all cliques is even. This would increase the period by one. The additional time unit will be allotted only to the dummy interval and thus can be ignored. We note that to produce the schedule at time t we just have to follow the recursive calls that include t in their period. Since there are no more than $\log T$ such calls, the time it takes to produce this schedule is polynomial in n for max-min fair frequencies.

LEMMA 4.2. *The algorithm produces a correct schedule for every feasible set of frequencies.*

LEMMA 4.3. *If the set of frequencies is of the form $1/2^i$ then the drift can be bounded by 1 and hence the response time can be bounded by $\lceil 2/f_i \rceil$.*

Proof: Since our algorithm always divides even s_i into equal halves, the following invariant is maintained: At any recursive level, whenever $s_i > 1$, then s_i is even. Also note that $T = 2^k$, where $\min_i f_i = 1/2^k$ and thus we can express each f_i as $2^{\psi_i - k}$. Now, following the algorithm, it can be easily shown that there is at least one occurrence of task i in each time interval of size $2^{k-\psi_i}$. Hence $\sum_{r=1}^{t} S(i,r) = \lfloor \frac{t}{2^{k-\psi_i}} \rfloor$ and P-fairness follows. □

LEMMA 4.4. *The response time for every interval I_i is bounded by $\lceil 4/f_i \rceil$.*

Proof: The proof is based on the Lemma 4.3. This lemma clearly implies the case in which the frequencies are powers of two. Moreover, in case the frequencies are not powers of two, we can virtually partition each task into two tasks with frequencies p_i and r_i respectively, so that $f_i = p_i + r_i$, p_i is a power of two, and $r_i < p_i$. Then, the schedule of the task with frequency p_i has drift 1. This implies that its response time is $\lceil 2/p_i \rceil \leq \lceil 4/f_i \rceil$. □

We remark that it can be shown that the bound of the above lemma is tight for our algorithm.

We summarize the results in this section in the following theorem:

THEOREM 4.5. *Given an arbitrary interval graph as conflict graph, the algorithm exactly realizes any feasible frequency-vector and guarantees that $r_i \leq \lceil 4/f_i \rceil$.*

4.1 Bounding the drift

Since the algorithm has $O(\log T)$ levels of recursion and each level may increase the drift by one, clearly the maximum drift is bounded by $O(\log T)$. In this section we prove that we can decrease the maximum drift to be $O(\sqrt{\log T} n^{\epsilon})$, for any fixed ϵ, where n is the number of tasks. By Theorem 2.6 this implies that in the worst case the drift for a max-min fair frequencies is bounded by $O(n^{\frac{1}{2}+\epsilon})$.

Our method to get a better drift is based on the following observation: At each recursive step of the algorithm two sets of tasks are produced such that each set has to be placed in a different half of the time-interval currently considered. However, we are free to choose which set goes to which half. We are using this degree of freedom to decrease the drift. To make the presentation clearer we assume that T is a power of two and that the time units are $0, \ldots, T-1$.

Consider a sub-interval of size $T/2^j$ starting *after* time $t_\ell = i \cdot T/2^j - 1$ and ending at $t_r = (i+1) \cdot T/2^j - 1$, for $0 \leq i \leq 2^j - 1$. In the first j recursion levels we already fixed the number of occurrences of each task up to t_ℓ. Given this number, the drift d_ℓ at time t_ℓ is fixed. Similarly, the drift d_r at time t_r is also fixed. At the next recursion level we split the occurrences assigned to the interval $[t_\ell + 1, t_r]$, and thus fixing the drift d_m at time $t_m = (t_\ell + t_r)/2$. Optimally, we would like the drifts after the next recursion level at each time unit $t \in [t_\ell + 1, t_r]$ to be the weighted average of the drifts d_ℓ and d_r. In other words, let $\alpha = (t - t_\ell)/(t_r - t_\ell)$, then, we would like the drift at time t to be $\alpha d_r + (1 - \alpha) d_\ell$. In particular, we would like the drift at t_m to be $(d_\ell + d_r)/2$. This drift can be achieved for t_m only if the occurrences in the interval $[t_\ell + 1, t_r]$ can be split equally. However, in case we have an odd number of occurrences to split, the drift at t_m is $(d_\ell + d_r)/2 \pm 1/2$, depending on our decision in which half interval to put the extra occurrence. Note that the weighted average of the drifts of all other points changes accordingly. That is, if the new d_m is $(d_\ell + d_r)/2 + x$, for $x \in \{\pm 1/2\}$, then the weighted average in $t \in [t_\ell + 1, (t_r + t_\ell)/2]$ is $\alpha d_r + (1-\alpha) d_\ell + 2\alpha x$, where $\alpha = (t - t_\ell)/(t_r - t_\ell) \leq 1/2$, and the weighted average in $t \in [(t_r + t_\ell)/2 + 1, t_r]$ is $\alpha d_r + (1-\alpha) d_\ell + (2 - 2\alpha) x$, where $\alpha = (t - t_\ell)/(t_r - t_\ell) > 1/2$.

Consider now the two sets of tasks S_1 and S_2 that we have to assign to the two sub-intervals (of the same size) at level k of the recursion. For each of the possible two assignments, we compute a "potential" based on the resulting drifts at time t_m. For a given possibility let $D[t_m, i, k]$ denote the resulting drift of the i-th task at t_m after k recursion levels. Define the potential of t_m after k levels as $POT(t_m, k) = \sum_{i=1}^{n} D(t_m, i, j)^{\phi}$, for some fixed even constant ϕ. We choose the possibility with the lowest potential.

THEOREM 4.6. *Using the policy described above the maximum drift is bounded by $O(\sqrt{\log T} \cdot n^{\epsilon})$, for any fixed ϵ.*

5 Realizing frequencies rigidly

In this section we show how to construct a schedule that 12-approximates any feasible frequency-vector in a *rigid* fashion on an interval graph. We reduce our Rigid Schedule problem to the Dynamic Storage Allocation problem. The Dynamic Storage Allocation

problem is defined as follows. We are given objects to be stored in a computer memory. Each object has two parameters: (i) its size in terms of number of cells needed to store it, (ii) the time interval in which it should be stored. Each object must be stored in adjacent cells. The problem is to find the minimal size memory that can accommodate at any given time all of the objects that are needed to be stored at that time. The Dynamic Storage Allocation problem is a special case of the multi-coloring problem on intervals graphs which we now define.

A *multi-coloring* of a weighted graph G with the weight function $w : V \to \mathcal{N}$, is a function $F : V \to 2^{\mathcal{N}}$ such that for all $v \in V$ the size of $F(v)$ is $w(v)$, and such that if $(v, u) \in E$ then $F(v) \cap F(u) = \emptyset$. The multi-coloring problem is to find a multi-coloring with minimal number of colors. This problem is known to be an NP-Hard problem [10].

Two interesting special cases of the Multi-Coloring problem are when the colors of a vertex either must be adjacent or must be "spread well" among all colors. We call the first case the AMC problem and the second case the CMC problem. More formally, in a solution to AMC if $F(u) = \{x_1 < \cdots < x_k\}$, then $x_{i+1} = x_i + 1$ for all $1 \leq i < k$. Whereas in a solution to CMC which uses T colors, if $F(u) = \{x_1 < \cdots < x_k\}$ then (i) k divides T, and (ii) $x_{i+1} = x_i + T/k$ for all $1 \leq i < k$, and $x_k + T/k - T = x_1$.

It is not hard to verify that for interval graphs the AMC problem is equivalent to the Dynamic Storage Allocation problem described above. Simply associate each object with a vertex in the graph and give it a weight equal to the number of cells it requires. Put an edge between two vertices if their time intervals intersect. The colors assigned to a vertex are interpreted as the cells in which the object is stored.

On the other hand, the CMC problem corresponds to the Rigid Schedule problem as follows. First, we replace the frequency $f(v)$ by a weight $w(v)$. Let $k(v) = \lceil -\log_2 f(v) \rceil$, and let $k = \max_{v \in V}\{k(v)\}$, then $w(v) = 2^{k-k(v)}$. Clearly, $w(v)/2^k \geq f(v)/2$.

Now, assume that the output for the CMC problem uses T colors and let the colors of v be $\{x_1 < \cdots < x_k\}$ where $x_2 - x_1 = \Delta$. We interpret this as follows: v is scheduled in times $x_1 + i\Delta$ for all $i \geq 0$. It is not difficult to verify that this is indeed a solution to the Rigid Scheduling problem.

Although Dynamic Storage Allocation problem is a special case of the multi-coloring problem it is still known to be an NP-Hard problem [10] and for similar reasons the Rigid Scheduling problem is also NP-Hard. Therefore, we are looking for an approximation algorithm. In what follows we present an approximation algorithm that produces a rigid scheduling that 12-approximates the given frequencies. For this we consider instances of the AMC and CMC problems in which the input weights are powers of two.

DEFINITION 5.1. *A solution for an instance of AMC is both* aligned *and* contiguous *if for all* $v \in V$ $F(v) = \{j \cdot w(v), \ldots, (j+1) \cdot w(v) - 1\}$ *for some* $j \geq 0$.

In [15], Kierstead presents an algorithm for AMC that has an approximation factor 3. A careful inspection of this algorithm shows that it produces solutions that are both aligned and contiguous for all instances in which the weights are power of two.

We show how to translate a solution for such an instance of the AMC problem that is both aligned and contiguous into a solution for an instance of the CMC problem with the same input weights.

For $0 \leq x < 2^k$, let $\pi(x)$ be the k-bit number whose binary representation is the inverse of the binary representation of x.

LEMMA 5.1. *For* $1 \leq i \leq k$ *and* $0 \leq j < 2^{k-i} = \Delta$, $\{\pi(j2^i), \ldots, \pi(j2^i + 2^i - 1)\} = \{\pi(j2^i), \pi(j2^i) + \Delta, \ldots, \pi(j2^i) + (2^i - 1)\Delta\}$.

Consider an instance of the CMC problem in which all the input weights are powers of two. Apply the solution of Kierstead [15] to solve the AMC instance with the same input. This solution is both aligned and contiguous, and uses at most $3T'$ colors where T' is the number of colors needed by an optimal coloring. Let $T \geq 3T'$ be the smallest power of 2 that is greater than T'. It follows that $T \leq 6T'$. Applying the transformation of Lemma 5.1 on the output of the solution to AMC yields a solution to CMC with at most T colors. This in turn, yields an approximation factor of at most 12 for the Rigid Scheduling problem, since $w(v)/T \geq f(v)/2$.

THEOREM 5.2. *The above algorithm computes a rigid schedule that 12-approximates any feasible frequency-vector on an interval graph.*

6 Circular-Arc graphs

In this section we show how to transform any algorithm $\mathcal{A}$ for computing a schedule that c-approximates any given feasible frequency-vector on interval graphs into an algorithm $\mathcal{A}'$ for computing a schedule that $2c$-approximates any given feasible frequencies on circular-arc graphs.

Let $\hat{f}$ be a feasible frequency-vector on a circular-arc graph G.

Step 1: Find the maximum clique C in G.

Let $G' = G - C$. Note that G' is an interval graph. Let $\hat{g_1}$ and $\hat{g_2}$ be the frequency-vectors resulting from restricting $\hat{f}$ to the vertices of G' and C, respectively. Note that $\hat{g_1}$ and $\hat{g_2}$ are feasible on G' and C, respectively.

Step 2: Using $\mathcal{A}$, find schedules S_1 and S_2 that c-approximate $\hat{g_1}$ and $\hat{g_2}$ on G' and C, respectively.

Step 3: Interleave S_1 and S_2.

Clearly, the resulting schedule $2c$-approximates $\hat{f}$ on the circular-arc graph G.

7 Future research

Many open problems remain. The exact complexity of computing a max-min fair frequency assignment in general graphs is not known and there is no characterization of when such an assignment is easy to compute. All the scheduling algorithms in the paper use the inherent linearity of interval or circular-arc graphs. It would be interesting to find scheduling algorithms for the wider class of perfect graphs. The algorithm for interval graphs that realizes frequencies exactly exhibits a considerable gap in its drift. It is not clear from which direction this gap can be closed.

Our algorithms assume a central scheduler that makes all the decisions. Both from theoretical and practical point of view it is important to design scheduling algorithms working in more realistic environments such as high-speed local-area networks and wireless networks (as mentioned in Section 1.1). The distinguishing requirements in such an environment include a distributed implementation via a local signaling scheme, a conflict graph which may change with time, and restrictions on space per node and size of a signal. The performance measures and general setting, however, remain the same. A first step towards such algorithms has been recently carried out by Mayer, Ofek and Yung in [19].

Acknowledgment. We would like to thank Don Coppersmith and Moti Yung for many useful discussions.

References

[1] B. Awerbuch and M. Saks, A Dining Philosophers Algorithm with Polynomial Response Time. *Proc. 31st IEEE Symp. on Foundations of Computer Science* (1990), 65–75.

[2] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, Proportionate Progress: A Notion of Fairness in Resource Allocation. *Proc. 25th ACM Symp. on Theory of Computing* (1993), 345–354.

[3] D. Bertsekas and R. Gallager, Data Networks. *Prentice Hall* (1987).

[4] J. Bar-Ilan and D. Peleg, Distributed Resource Allocation Algorithms. *Proc. 6th International Workshop on Distributed Algorithms* (1992), 277–291.

[5] J. Chen, I. Cidon, and Y. Ofek, A Local Fairness Algorithm for Gigabit LANs/MANs with Spatial Reuse. *IEEE J. on Selected Areas in Communication*, 11(8):1183–1192 (1993).

[6] K. Chandy and J. Misra, The Drinking Philosophers Problem. *ACM Trans. on Programming Languages and Systems*, 6:632–646 (1984).

[7] I. Cidon and Y. Ofek, MetaRing – A Full-Duplex Ring with Fairness and Spatial Reuse. *IEEE Trans. on Communications*, 41(1):110–120 (1993).

[8] M. Choy and A. Singh, Efficient Fault Tolerant Algorithms for Resource Allocation in Distributed System. *Proc. 24th ACM Symp. on Theory of Computing* (1992), 593–602.

[9] E. W. Dijkstra, Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1:115–138 (1971).

[10] M. Garey and D. Johnson, Computers and Intractability, a Guide to the Theory of NP-Completeness, *W. H. Freeman*, San Francisco, 1979.

[11] M. Golumbic, Algorithmic Graph Theory and Perfect Graphs. *Acadmeic Press*, New York, 1980.

[12] M. Grötschel, L. Lóvasz and A.Schrijver, Geometric Algorithms and Combinatorial Optimization. *Springer-Verlag*, Berlin, 1987.

[13] D. J. Goodman, Cellular Packet Communications. *IEEE Trans. on Communications*, 38:1272–1280 (1990).

[14] J. Jaffe, Bottleneck Flow Control. *IEEE Trans. on Communications*, 29(7):954–962 (1981).

[15] H. A. Kierstead, A Polynomial Time Approximation Algorithm for Dynamic Storage Allocation. *Discrete Mathematics*, 88:231–237 (1991).

[16] D. E. Knuth, The Sandwich Theorem, *The Electronic Journal of Combinatorics*, 1:1–48, (1994).

[17] C. Lund and M. Yannakakis, On the Hardness of Approximating Minimization Problems. *Proc. 25th ACM Symp. on Theory of Computing* (1993), 286–293.

[18] N. Lynch, Fast Allocation of Nearby Resources in a Distributed System. *Proc. 12th ACM Symp. on Theory of Computing* (1980), 70–81.

[19] A. Mayer, Y. Ofek, and M. Yung, Distributed Scheduling Algorithm for Fairness with Minimum Delay. *To be submitted to ACM Sigcomm'95.*

[20] A. Tucker, Matrix characterizations of circular-arc graphs. *Pacific Journal of Mathematics*, 39:535–545, (1971).