

Efficient Periodic Scheduling by Trees

Amotz Bar-Noy

amotz@sci.brooklyn.cuny.edu

Computer and Information Science Department

Brooklyn College

2900 Bedford Avenue

Brooklyn, NY 11210

USA

Vladimir Dreizin

vld@eng.tau.ac.il

Boaz Patt-Shamir

boaz@eng.tau.ac.il

Dept. of Electrical Engineering

Tel Aviv University

Ramat Aviv

Tel Aviv 69978

Israel

Abstract—In a *perfectly-periodic schedule*, time is divided into time-slots, and each client gets a time slot precisely every predefined number of time slots. The input to a schedule design algorithm is a frequency request for each client, and its task is to construct a perfectly periodic schedule that matches the requests as “closely” as possible. The quality of the schedule is measured by the ratios between the requested frequency and the allocated frequency for each client (either by the weighted average or by the maximum of these ratios over all clients). Periodic schedules enjoy maximal fairness, and are very useful in many contexts of asymmetric communication, e.g., push systems and Bluetooth networks. However, finding an optimal periodic schedule is NP-hard in general. *Tree scheduling* is a methodology for developing perfectly periodic schedules with quality guarantees by constructing trees that correspond to periodic schedules. We explore a few aspects of tree scheduling. First, noting that a complete schedule table may be exponential in size, and that using the tree for scheduling directly may require logarithmic time on average, we give algorithms that find the next client to schedule in constant amortized time, using only polynomial space in most practical cases. Second, we present a few heuristic algorithms for generating schedules, based on analysis of optimal tree-scheduling algorithms, for both the average and maximum measures. Simulation results indicate that some of these heuristics produce excellent schedules in practice, sometimes even beating the best known non-periodic schedules.

Index Terms—periodic schedules, fair scheduling, broadcast disks, Bluetooth, push systems

I. INTRODUCTION

One of the major problems of mobile communication devices is power supply, partly due to the fact that radio communication is a relatively high power consumer. A common way to mitigate this difficulty is to use scheduling strategies that allow mobile devices to keep their radios turned off for most of the time. This approach can be found in many layers, from the data-link layer to the application layer. For example, Bluetooth’s “Park Mode” and “Sniff Mode” allow a client to sleep except for some predefined periodic interval [10]. Another example is Broadcast Disks [1], where a server broadcasts “pages” to clients. The goal is to minimize the waiting time and, in particular, the “busy waiting” time of a random client that wishes to access one of the pages [15].

One class of particularly attractive schedules (from the client’s point of view) is the class of *perfectly periodic* schedules, where each client i gets one time slot exactly every β_i

time slots, for some β_i called the *period* of i . Under a perfectly periodic schedule, a client needs to record only two numbers (period length and offset) to get a full description of its own schedule. It is important to observe that periodic schedules have, in some sense, the “best” fairness among all schedules (cf., for example, the “chairperson assignment problem” [17]). In Broadcast Disks, other schedules that only guarantee low waiting time may require the client to actively listen until its turn arrives (busy-waiting), while perfectly periodic schedules allow the client to actually shut down its receiver.

The main questions for periodic scheduling are how to find good schedules, and how to implement them on the server’s side. More specifically, the model is roughly as follows. We assume that we are given a set of *share requests* $\{a_1, \dots, a_n\}$ where each a_i represents the fraction of the bandwidth requested by client i , i.e., $\sum_{i=1}^n a_i = 1$. Given this input, an algorithm computes a perfectly periodic schedule that matches the clients requests as “closely” as possible. The schedule implies a period β_i and a share $b_i = 1/\beta_i$ for each client i . Measuring the goodness of a schedule is done based on the ratio of the granted shares b_i to the requested shares a_i : it makes sense, depending on the target application, to be concerned either by the weighted average of these ratios, or by the maximum of these ratios. Formally, we define $\rho_i = \frac{a_i}{b_i}$ for each i , the ratio of the requested share to the granted share. Using the ρ_i ’s we define the performance measures:

Maximum: $\text{MAX} = \max \{\rho_i \mid 1 \leq i \leq n\}$.

Weighted Average: $\text{AVE} = \sum_{i=1}^n a_i \rho_i$.

Unfortunately, there are serious difficulties in using periodic schedules. First, it is known that finding an optimal schedule (under either the maximum or average measure) is NP-hard in general [6]. And second, even when the schedule is given somehow, it is not clear how to represent it in a way that allows for efficient usage at the server, since the length of a full cycle of a perfectly periodic schedule may be exponential in the number of clients, or in the longest period (as we show in this paper).

In this paper, we study perfectly periodic schedules from the *tree scheduling* point of view [8]. In tree scheduling, the idea is to take advantage of the natural correspondence between trees and perfectly periodic schedules: leaves correspond to clients, and the period of each client is the product of the degrees of the nodes on the path leading from the root to its corresponding leaf

Part of the work was done while the first author was with Tel Aviv University and AT&T Research Labs.

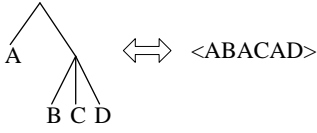


Fig. 1. An example of a tree and its corresponding schedule.

(see example in Figure 1; a more detailed explanation is given in Section III). Our results address two questions.

First, we look at the implementation issue. We observe that the main problem here is that a naïve implementation of a schedule, where the schedule is represented simply by listing its full cycle, may be very expensive in terms of space: the cycle length may be as high as $\Omega(2^n)$ for n clients. Note that finding the next client to schedule, however, is just one table access. On the other hand, if we use a tree to represent the schedule, the space requirement drops to $O(n)$, but now even the *average* time to find the next client to schedule is, on some trees, as high as $\Omega(\log n)$. Based on these observations, we give an algorithm that combines the benefits of both methods, finding the next client to schedule in $O(1)$ time on average for all trees, using only polynomial space in most practical cases.

Next, we address the question of finding trees that represent good periodic schedules. Our strategy here is to analyze the optimal (exponential time) algorithm for tree scheduling, and using it as the starting point, we develop a few heuristic algorithms that run in polynomial time. In extensive tests we ran, we found that our best algorithm manages to beat even the best known non-periodic algorithm in practical circumstances. This is interesting, since perfect periodicity is not always possible (consider, for example, the requests $a_1 = 1/2$ and $a_2 = 1/3$).

We believe that these results help validate the claim that periodic scheduling in general, and tree scheduling in particular, are viable approaches for scheduling.

A. Related work

Motivated by the goal of minimizing the waiting time, much research has focused on scheduling which is not perfectly periodic, using the average measure as the target function. For example, [6] presents an algorithm that produces a schedule whose average measure is at most $\frac{9}{8}$ times the best possible. That algorithm uses the golden ratio schedule, and hence gaps between consecutive occurrences of a client have three distinct values. In [14], Kenyon *et al.* describe a polynomial approximation scheme for the average measure; their solution is not perfectly periodic either.

Hameed and Vaidya [19] propose using Weighted Fair Queuing to schedule broadcasts (which results in non-periodic schedules). They made the distinction between the schedule design stage and the on-line scheduling task; their on-line scheduling algorithm takes $O(\log n)$ steps per time slot on average.

Ammar and Wong [4], [5], motivated by Teletext systems, show that the optimal schedule is cyclic, and give a 2-approximation algorithm for periodic scheduling. Khanna and Zhou [15] show how to use indexing with periodic scheduling to minimize busy waiting, and they also show how to get a $\frac{3}{2}$ approximation w.r.t. the average measure. Bar-Noy *et al.* prove in

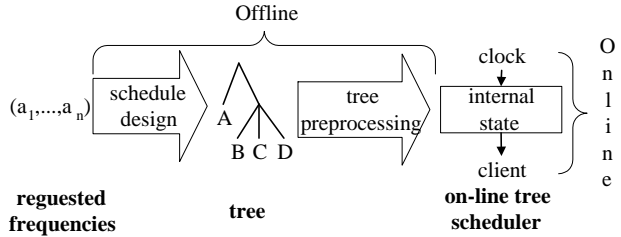


Fig. 2. Solution stages.

[6] that it is NP-hard to find the optimal perfect periodic schedule.

The work of [8] is the most relevant to the current paper. In [8], the notion of perfect periodicity is introduced, as well as the tree methodology. The main results in [8] are algorithms for tree schedule design whose resulting schedules are guaranteed to be close to optimum w.r.t. the average measure (between $\frac{4}{3}$ and $1 + \epsilon$, depending on the value of the maximal requested share).

Additional papers with analysis of the average measure (motivated by Broadcast Disks and related problems) are [1], [9], [16], [18]. The machine maintenance problem [20], [2] and the chairperson assignment problem [17] are also closely related to periodic scheduling.

B. Our contribution

In this paper we study some properties of periodic schedules in general, and of tree scheduling in particular.

- For general periodic schedules, we analyze the cycle length as a function of the number of clients or the length of the longest period.

Regarding tree scheduling, we view it as a process that consists of two tasks (see Figure 2): First, given a vector of share requests, the *schedule design* task is to find a tree that represents the schedule. The second task, which we call *on-line tree scheduling*, is to actually determine which client to schedule in every slot. To do that efficiently, schedule representation should be carefully designed. In this paper, we propose efficient solutions for both tasks: we give a few heuristic schedule design algorithms (based on optimal algorithms for tree scheduling), and we give an algorithm for efficient on-line scheduling.

- The on-line scheduling algorithm is the first to guarantee $O(1)$ amortized time complexity, using polynomial space (except for some pathological cases).
- The perfectly-periodic schedules produced by our heuristic algorithms on randomly generated data have comparable performance with the best *non-periodic* schedules, both for the max and the average measures.

Due to lack of space, many details and proofs are omitted. The interested reader is referred to [12].

II. DEFINITIONS AND PRELIMINARIES

A. Schedules

We are given a set of n clients $1, 2, \dots, n$. Each client i requests a share $0 < a_i < 1$ (of a common resource). We refer to

a_i as a *requested share* or a *frequency demand* of client i . A *frequency demand vector* is a vector $\mathcal{A} = \langle a_1, a_2, \dots, a_n \rangle$, where $\sum_{i=1}^n a_i = 1$. Sometimes we state client demands in terms of *requested periods* $\alpha_i = 1/a_i$.

A *schedule* is an infinite sequence $S = s_0, s_1, \dots$, where $s_j \in \{1, 2, \dots, n\}$ for all j . A schedule is *cyclic* if it is an infinite concatenation of a finite sequence $C = \langle s_0, s_1, \dots, s_{|C|-1} \rangle$. C is a *cycle* of S . In this paper we deal mainly with cyclic schedules, and we refer interchangeably to a schedule and to its cycle. A schedule is *perfectly periodic* (or just *perfect* for short) if slots allocated to each client are equally spaced, i.e. for each client i there exist non-negative integers $\beta_i, o_i \in \mathbb{Z}^+$ called the *period* and *offset* of i , respectively, such that i is scheduled in slot j if and only if $j \equiv o_i \pmod{\beta_i}$. The frequency b_i of a client i in a perfect schedule is the reciprocal of its period, i.e., $b_i = 1/\beta_i$. We refer to b_i as the *granted share* and to β_i as the *granted period* for client i .

For later reference, we state the following property of perfect schedules (recall that lcm denotes the least common multiple).

Lemma II.1: Let S be a perfect schedule with periods β_1, \dots, β_n . Then the minimal cycle length of S is $\text{lcm}(\beta_1, \dots, \beta_n)$.

The proof is based on the fact that the period of each client must divide the cycle length.

B. Measures

Given a frequency demand vector \mathcal{A} and a granted frequency vector \mathcal{B} we define the following measures:

$$\begin{aligned} \text{MAX}_{\mathcal{A}, \mathcal{B}} &= \max \{a_i/b_i \mid 1 \leq i \leq n\} \\ \text{AVE}_{\mathcal{A}, \mathcal{B}} &= \sum_{i=1}^n a_i^2/b_i \end{aligned}$$

We omit subscripts when they are clear by the context.

Intuitively, the “best” (or “fairest”) perfect schedule should be the one that provides each client exactly with its demand, i.e., $b_i = a_i$ for all i . In this case we get that $\text{MAX} = \text{AVE} = 1$. The following lemma states that this is indeed the best possible. The proof is based on elementary calculus, and is therefore omitted.

Lemma II.2: For all frequency vectors \mathcal{A}, \mathcal{B} , we have that $\text{MAX}_{\mathcal{A}, \mathcal{B}} \geq 1$ and $\text{AVE}_{\mathcal{A}, \mathcal{B}} \geq 1$.

We remark that for some other measures, providing each client with its demand is *not* the best possible: in [12] we show that this is the case for weighted maximum, and unweighted average.

Example. Let $a_1 = 1/2$, $a_2 = 1/3$, and $a_3 = 1/6$. We compare the *round robin* schedule $RR = \langle 1, 2, 3 \rangle$ with another perfect schedule with three clients $C = \langle 1, 2, 1, 3 \rangle$. The following table summarizes the performance of the two schedules for the two measures. The table shows that S outperforms RR in both measures.

Schedule	(β_1, ρ_1)	(β_2, ρ_2)	(β_3, ρ_3)	MAX	AVE
RR	$(3, 3/2)$	$(3, 1)$	$(3, 1/2)$	$3/2$	$7/6$
S	$(2, 1)$	$(4, 4/3)$	$(4, 2/3)$	$4/3$	$19/18$

Another example is when $a_1 = 1/3$, $a_2 = 1/3$, $a_3 = 1/4$, and $a_4 = 1/12$. We compare four perfect schedules with four clients: $RR = C_1 = \langle 1, 2, 3, 4 \rangle$, $CS_2 = \langle 1, 2, 3, 1, 2, 4 \rangle$, $CS_3 = \langle 1, 2, 1, 3, 1, 4 \rangle$, and $C_4 = \langle 1, 2, 1, 3, 1, 2, 1, 4 \rangle$. The reader may verify that the round-robin schedule C_1 is the best for the MAX measure, while C_2 is the best schedule for the AVE measure.

C. Trees

A *tree* is a connected acyclic graph. A *rooted tree* is a tree with one node designated as the *root*. We assume that all edges are directed away from the root. If (u, v) is a directed edge, then v is the *child* of u , and u is the *parent* of v . The *degree* of a node in a rooted tree is the number of its children. A *leaf* is a node with degree 0. An *ordered tree* is a rooted tree where each node has a total order on its children. The *level* of a node in a rooted tree is the length of the path from the root to that node. A *level-uniform tree* is a tree with a number d_j associated with level j , such that each node in level j is either a leaf or has degree d_j .

III. SCHEDULES, TREES AND TREE SCHEDULING

Tree scheduling is a methodology for constructing perfect schedules that are based on ordered trees [8]. The idea is simple: each leaf corresponds to a *distinct* client, and the period of each client is the product of the degree of all the nodes on the path leading from the root to the corresponding leaf. In the example of Figure 1, the period of A is 2 because the root degree is 2, and the periods of B, C and D are 6, because the root degree is 2 and the degree of their parent is 3. To get an explicit schedule, the offset of each client has to be computed as well (see details below). We refer to a tree that represents a schedule as a *schedule tree*. If T is a schedule tree, then $C(T)$ denotes its corresponding schedule cycle.

In this section, we show how to convert a schedule tree into a cycle of the schedule, and derive a few quantitative bounds on the size of the cycle.

A. Procedure offLine

The basic idea in constructing a schedule from a tree is to work recursively. We use the interleave operator to merge schedules from different subtrees.

The interleave operator. Let $C_1 = \langle s_1^1, s_2^1, \dots, s_\ell^1 \rangle, \dots, C_k = \langle s_1^k, s_2^k, \dots, s_\ell^k \rangle$ be equal-length cycles of periodic schedules. The interleave operator is defined by taking one client from each schedule in a round-robin fashion. Formally:

$$\begin{aligned} \text{interleave}(C_1, \dots, C_k) &= \\ &\langle s_1^1, \dots, s_1^k, s_2^1, \dots, s_2^k, \dots, s_\ell^1, \dots, s_\ell^k \rangle \end{aligned}$$

If the clients in the schedules C_1, \dots, C_k are mutually disjoint, and each C_i is perfect, then the resulting schedule is also perfect.

Procedure offLine gets an ordered tree and produces a cycle of the corresponding schedule. Pseudo code for the procedure named offLine is given in Figure 3. An example of an execution

Procedure offLine

Input: schedule tree T with leaves $1, \dots, n$
Output: cycle of perfect schedule C of length ℓ

if T is a single leaf i
 $C \leftarrow \langle i \rangle$

else
 Let T_0, \dots, T_{d-1} be the d subtrees of T
for all $0 \leq i < d$ **do**
 $C_i \leftarrow \text{offLine}(T_i)$
 $\ell_i \leftarrow |C_i|$
 $\ell' \leftarrow \text{lcm}(\ell_0, \dots, \ell_{d-1}); \ell \leftarrow d\ell'$
for all $0 \leq i < d$ **do**
 $C'_i \leftarrow$ concatenation of ℓ'/ℓ_i copies of C_i
 $C \leftarrow \text{interleave}(C'_0, \dots, C'_{d-1})$

Fig. 3. Procedure offLine.

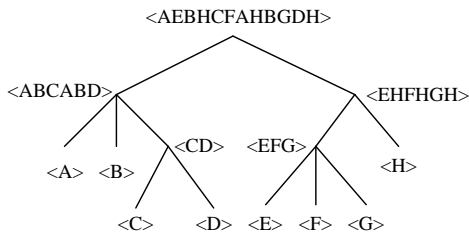


Fig. 4. Execution of offLine: each node is labeled with the intermediate schedule returned by a recursive call.

of offLine is illustrated in Figure 4, where each node is labeled with the schedule cycle corresponding to its subtree. We summarize its main properties below.

Lemma III.1: Let T be a tree with n leaves labeled $1, \dots, n$, and let $C(T)$ be the schedule cycle generated for T by offLine. Then $C(T)$ is perfectly periodic, where the period of each client i in $C(T)$ is the product of the degrees of all ancestors of i in T . Moreover, the output length is the lcm of the periods of all clients.

B. Properties of perfect tree schedules

We now turn to study the *size* of the cycles of tree schedules. We prove lower and upper bounds on the worst-case cycle length. The proofs are somewhat technical, and are omitted from this extended abstract. The results are used in the analysis of on-line scheduling in Section IV.

The first result bounds the length of the cycle in terms of the maximal degree in the tree and the maximal period in the schedule.

Theorem III.1: Let T be a schedule tree with schedule cycle C . Suppose that degrees in T are bounded by Δ , and that the periods of clients in C are bounded by β . Then $|C| \leq \beta^{\pi(\Delta)}$, where $\pi(\Delta) \approx \frac{\Delta}{\ln \Delta}$ is the number of prime numbers smaller than or equal to Δ .

We note that in general, the length of the schedule may be exponential in the maximal period length (Theorem III.3 below). Theorem III.1 gives a better upper bound in case that Δ is small. The next simple lemma deals with level-uniform trees.

Lemma III.2: Let T be a level-uniform schedule tree, with

corresponding schedule cycle C . Then $|C|$ is exactly the maximal period in schedule.

The next theorem shows an interesting bound on the cycle length for any tree schedule in terms of the number of leaves. In some sense, it shows that the example of Figure 5 is the worst possible, including all “crazy” trees.

Theorem III.2: Let T be a tree with n leaves, and let C be its corresponding tree schedule. Then $|C| \leq 2^{n-1}$.

The idea in the proof of Theorem III.2 is to show that for any tree T there exists a binary tree with the same number of leaves and larger schedule size. We prove it by converting a non-binary tree into a tree with less non-binary nodes and larger schedule size. The bound follows from the bound on binary trees.

The next theorem shows that the schedule length can be exponential also in the maximal period length, not only in the number of clients.

Theorem III.3: There exists a tree T with largest period B , such that its corresponding schedule size is $2^{\Omega(\sqrt{B/\ln \ln B})}$.

IV. ON-LINE TREE SCHEDULING

In this section we consider the on-line scheduling problem. Our starting point is the observation that simply listing a complete cycle of the schedule (as offLine does) may be too expensive: Consider, for example, the schedule where the periods are defined as follows:

$$\beta_i = \begin{cases} 2^i, & \text{for } 1 \leq i < n \\ 2^{n-1}, & \text{for } i = n \end{cases}$$

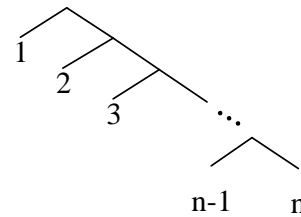


Fig. 5. The tree corresponding to a schedule of 2^{n-1} slots.

The tree corresponding to this schedule is depicted in Figure 5. The minimal cycle length for this schedule, by Lemma II.1, is 2^{n-1} . Clearly, this representation is wasteful: here is a space efficient solution. We maintain an $(n-1)$ -bit counter; in each time slot we increment it modulo 2^{n-1} , and we schedule client $i+1$ iff the i least significant bits of the counter are 0. It is straightforward to verify that this scheme works, but there is one significant drawback to it (besides the obvious question of generalization): while the cycle-listing approach is expensive in space, each scheduling step is one table access.¹ The second approach, whose space complexity is minimal ($O(n)$, proportional to the number of clients), is expensive in terms of worst-case processing time: in some cases, the number of comparisons required to determine the identity of the next client to schedule is $\Omega(n)$.

¹Asymptotically, a table access may take more than one time unit. See Remark 1 at the end of the section.

In this section, we first show an inherent lower bound on the worst-case time complexity of a single invocation of on-line scheduling. In light of this result, we resort to good average time complexity: We present an on-line algorithm and show that its average running time is constant for all schedules.

A. Worst-case running time of on-line scheduling

The theorem below shows that the worst-case time complexity of on-line scheduling, for any schedule, is at least logarithmic in the cycle length.

Theorem IV.1: Let S be a schedule with cycle C , and suppose that in each time unit, at most w bits can be tested. Then the worst-case running time of any on-line scheduling algorithm is $\Omega(\log |C|/w)$.

The proof is based on the Branching Program computational model [7], and its crux is that if the total number of steps is at most T , then the total number of possible outputs is less than 2^{wT} .

Using Theorems IV.1, III.3 and the tree from Figure 5, we get the following results.

Corollary IV.1: Let w be the maximal number of bits that can be tested in a single time unit. Then:

- 1) There exists a tree schedule with n clients and worst-case on-line scheduling time $\Omega(n/w)$.
- 2) There exists a tree schedule with maximal period B and worst-case running time $\Omega(\sqrt{B}/\ln \ln B/w)$.

B. Space-efficient On-line tree scheduling

Procedure onLine

Input: A node $v \in T$

Output: A client c

Persistent state: Token placement on edges of T

Invariant: For each non-leaf node, exactly one token on one of the outgoing edges

Code:

while v is not a leaf **do**

 Let u_0, \dots, u_{d-1} be the d children of v

 Let (v, u_i) be the outgoing edge of v with token

 Move token to edge $(v, u_{(i+1) \bmod d})$.

$v \leftarrow u_i$

(* v is a leaf *)

return client corresponding to v

Fig. 6. Procedure onLine

We now describe a space-efficient algorithm called onLine for on-line tree scheduling. Pseudo code is given in Figure 6. The idea is to use the tree representation of the schedule and to compute each time the identity of the next client to schedule, based on *tokens* placed on tree edges. Specifically, each non-leaf node has exactly one token placed on one of the edges leading to its children. The algorithm, called each time slot, descends the tree starting from the root, by following the edges with tokens. In addition, each time an edge (u, v) is crossed, the token is moved to the edge leading to the next child of u , where “next” means using a cyclical order. When a leaf is reached, it is output as the next client to schedule.

We remark that the initial token placement and the way the cyclical orderings are defined on the edges outgoing from each node are immaterial to the correctness of the algorithm: the initial placement just determines the point in which the cycle starts, and the edge orderings “transpose” sub-schedules of equal weight.

We summarize the straightforward properties of onLine in the following theorem.

Theorem IV.2: Let T be a schedule tree whose corresponding cycle is C . Then $|C|$ applications of onLine procedure produce C . The worst-case running time of onLine is proportional to the height of T , and the space requirement of onLine is proportional to the size of the tree.

The more interesting measure of onLine is its *average* time complexity. We need the following definition.

Definition IV.1: Let T be a schedule tree with clients $1, \dots, n$ whose shares are b_1, \dots, b_n , respectively. The *entropy* of T , denoted $H(T)$, is the information-theoretic entropy of a source of symbols $1, \dots, n$ whose respective probabilities are b_i . Formally:

$$H(T) = \sum_{i=1}^n b_i \log \frac{1}{b_i}.$$

Clearly, $H(T)$ is exactly the average path length taken from the root of T to a random leaf, where leaf i is chosen with probability b_i . In other words, we have:

Theorem IV.3: Let T be a schedule tree. The total running time of $|C(T)|$ consecutive applications of onLine is $O(|C(T)| \cdot H(T))$.

Corollary IV.2: The amortized running time of the onLine procedure on any schedule tree with n leaves is $O(\log n)$.

Remarks.

- 1) For a rigorous treatment of time complexity, we must use the *bit complexity* model [3], where in each time unit only a bounded number of bits can be read and written. In this model, the worst-case access time is logarithmic in the space, so the full cycle listing algorithm may take $\Omega(n)$ time units for a single output if the cycle length is $\Omega(2^n)$. However, since memory accesses are usually implemented in hardware, we ignore this subtlety whenever possible.
- 2) Another implementation of on-line scheduling for perfect schedules uses a heap containing all clients. Each client is inserted with its next slot number that can be computed based on its period. The initial heap is based on the offsets of the clients. To keep space complexity under control, the numbers are reduced when their minimum is large enough. The space complexity of this algorithm is $O(n)$, and its worst-case running time seems to be $O(\log n)$ steps in the worst case. However, from the complexity theoretic point of view, the numbers that this algorithm manipulates may be as large as $|C|$, and hence each “step” actually takes $\Omega(\log |C|)$ time units in the worst case.

More importantly, the heap-based algorithm is not suited for the extensions we present next.

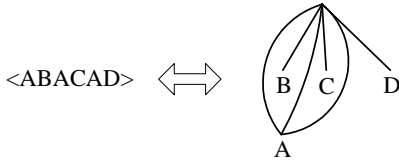


Fig. 7. A schedule represented by a round-robin dag.

C. A preprocessing procedure: Scheduling Dags

We now show how to combine `offLine` and `onLine` methods so that the amortized running time is reduced to constant. The intuition is that, at least for binary trees, since trees that are “bad” for `onLine` are trees whose entropy is high, these trees are shallow and therefore have small cycle length. On the other hand, trees with large cycle length are generally “skewed” (e.g., Figure 5) and have low entropy. The algorithm therefore combines `onLine` with `offLine` to get a small-space representation with low average time complexity.

Our first step is to extend the notion of schedule trees to *schedule dags*, by allowing more than one incoming edge per node, which means that there can be more than one path from the root to a leaf (see Fig. 7 for an example.) Formally, the scheduling graph is required to be a directed acyclic graph with exactly one root (a node without any incoming edge). Note that algorithms `offLine` and `onLine` can be applied without any change to a schedule dag. One important difference is that unlike schedule trees, not every schedule dag corresponds to a perfectly periodic schedule.

The main tools we use in the algorithm below are the functions `coalesce`, `expand`, and `rrDag`:

- The function $T' = \text{coalesce}(T, V)$ receives a tree T and a set of nodes V in T , and returns a new tree T' in which each subtree of T rooted at a node $v \in V$ is coalesced into a single *compound leaf*. We require that no node in V is a descendent of any other node in V .
- The function $T = \text{expand}(T', V)$ is the inverse of `coalesce`: it receives a tree T' and a set of compound leaves V in T' , and returns the tree T which results from expanding each compound leaf in V back to its original subtree.
- The function $T = \text{rrDag}(C)$ converts a listing of cycle C to a corresponding *round robin dag* as follows: First, a node v_i is introduced for each *distinct* entry i in the schedule, and a root node r is also introduced. Then, r is connected to each v_i according to the schedule list: the first edge is connected to the node corresponding to the first entry etc. (see Figure 7 for an example). Note that if a client appears more than once in the schedule list, then there is more than one edge to it from the root.

The idea in the preprocessing algorithm called `prepTree` we present next is the following (see pseudo code in Figure 8). Starting from the schedule tree, we coalesce nodes whose share is less than $1/n$ into compound nodes (a share of a node is the sum of shares of the leafs in its subtree). The coalesced tree is submitted to `offLine`, which returns a listing of the cycle. (Note that some of the entries in this cycle correspond to compound nodes.) This list is then converted to a round robin dag. Finally,

the compound nodes in the round robin dag are expanded back into trees (see Figure 9).

Algorithm `prepTree`

Input: A schedule tree T

Output: A schedule dag

$V \leftarrow \{t \in T \mid \beta_t \geq n \text{ and } \beta_{\text{parent}(t)} < n\}$.

$T_1 \leftarrow \text{coalesce}(T, V)$

$T_2 \leftarrow \text{rrDag}(\text{offLine}(T_1))$

return $\text{expand}(T_2, V)$

Fig. 8. Preprocessing procedure: gets a schedule tree and outputs a schedule dag with $O(1)$ amortized running time.

It is not hard to verify that applying `onLine` to the final schedule dag yields the same schedule, as we formally claim in the following lemma.

Lemma IV.1: Let T be a schedule tree, V a set of nodes of T , $T_1 = \text{coalesce}(T, V)$, $T_2 = \text{rrDag}(\text{offLine}(T_1))$, and $T' = \text{expand}(T_2, V)$. Then procedure `onLine` produces the same schedule given either T or T' .

We now analyze the average running time of `onLine` when applied to scheduling dags generated by `prepTree`. We will use the following technical result.

Lemma IV.2: For all $m, n \in \mathbb{N}$ and all $n_1, \dots, n_m \geq 0$ s.t. $\sum_{i=1}^m n_i = n$, we have $\sum_{i=1}^m \log n_i < \frac{n}{e \ln 2} \approx 0.53n$.

Theorem IV.4: The amortized running time of the `onLine` procedure on `prepTree`(T) is $O(1)$ for any schedule tree T .

Proof: Let V be the set of nodes in level 1 in the schedule dag that have children., i.e., the nodes that were coalesced by `prepTree`. Consider the scheduling of a client i . In the first step, the algorithm follows one edge in the schedule dag and reaches a node v . The next step depends on whether $v \in V$ or not. If $v \notin V$, the algorithm terminates by with output v . If $v \in V$, the algorithm descends T_v , where T_v denotes the subtree rooted at v . The crucial observations are that (1) the average running time of the algorithm on T_v is $H(T_v)$, the entropy of T_v , and hence it is at most $\log L(T_v)$, where $L(T_v)$ is the number of leaves in T_v ; and (2), by the specification of `prepTree`, we descend a tree only if the probability to reach its root is very small to begin

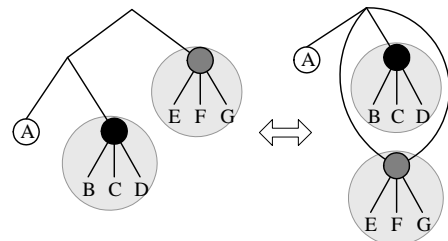


Fig. 9. Example of a schedule dag.

with. We can now make the calculations, using Lemma IV.2:

$$\begin{aligned}
\text{average time} &= 1 + \sum_{v \in V} \Pr[v \text{ is chosen}] \cdot H(T_v) \\
&\leq 1 + \frac{1}{n} \sum_{v \in V} \log L(T_v) \\
&\leq 1 + \frac{1}{e \ln 2} \approx 1.53.
\end{aligned}$$

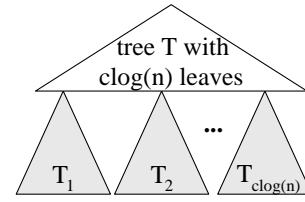


Fig. 10. If $T_1, \dots, T_{c \log n}$ are *good* trees, the overall tree is *good* too.

While the average running time of onLine is guaranteed to be constant on dags generated by prepTree, the size of these dags is not always small. However, we argue that for most practical cases, the space requirement is polynomial. Let us now list a few provably low space complexity cases.

Call a schedule tree *good* if it can be converted to schedule dag, for which the algorithm onLine runs in amortized constant time using polynomial space. The following types of trees are *good*.

All nodes have maximal degree Δ . In this case, the space complexity of the reorganization is polynomial in n by Lemma III.1. To get the schedule, one invocation of prepTree suffices.

Level-uniform trees. In this case, the space complexity of the reorganization is polynomial by Lemma III.2, again just by calling prepTree.

A tree T that consists of a tree with $O(\log n)$ leaves, on which *good* subtrees are hung (see Figure 10). For this case, we propose the following algorithm. Let $k = O(\log n)$ and let T_1, \dots, T_k be the *good* subtrees.

- 1) $T'_i \leftarrow \text{prepTree}(T_i)$ for $i = 1, \dots, k$.
- 2) replace T_i in T with T'_i for $i = 1, \dots, k$.
- 3) $T' \leftarrow \text{coalesce}(T, \{T'_1, \dots, T'_k\})$.
- 4) $T'' \leftarrow \text{rrDag}(\text{offLine}(T'))$.
- 5) return $\text{expand}(T'', \{T'_1, \dots, T'_k\})$.

The length of the schedule produced by offLine is polynomial by Lemma III.2. The space complexity required by the first stage is also polynomial since the T_i 's are *good* trees. Hence, the total space complexity is polynomial. The amortized running time of algorithm onLine on the resulting dag is still $O(1)$, since we only add one extra step from its root to the roots of the T'_i s.

A tree that consists of a constant number of *good* trees. Consider a tree consisting of a constant number of *good* trees, which are linked in the following manner: The root of one tree is a leaf of another. Our algorithm is to perform a reorganization of each *good* tree separately and then to connect the resulting schedule dags. It is obvious that the running time of the on-line algorithm is constant (since any root-leaf path consists of a constant number of dags, and crossing each dag takes constant time one average by assumption the the subtrees are *good*). The total space complexity is polynomial since the reorganization of each one of *good* trees is polynomial.

One useful example of such a tree is the “round-robin” tree: a node, on which k *good* subtrees T_1, \dots, T_k are hung. In this case we just reorganize T_1, \dots, T_k and link the resulting schedule dags back to the root node.

V. SCHEDULE DESIGN ALGORITHMS

In this section we describe algorithms to find schedule trees. We start with optimal (exponential time) algorithms for the MAX and AVE measures, and then develop polynomial time heuristic algorithms.

A. The optimal tree algorithms

Our algorithms use the *bottom-up* approach. That is, the algorithm combines a set of leaves into a single node and continues recursively. This approach is very similar to the construction of Huffman codes [13].

Our algorithms are based on the claim that for both MAX and AVE measures, it is always better to give more slots in the schedule for clients whose shares are larger. This means that there exists an optimal tree where the smallest shares in the schedule belongs to the client with the smallest share demand. In the following lemma, α_1 and α_2 are arbitrary requested periods and β and β' are the granted periods in the optimal schedule respectively. The first part of the lemma is for the MAX measure and the second part is for the AVE measure.

Lemma V.1: For $\alpha_1 \leq \alpha_2$ and $\beta \leq \beta'$,

- 1) $\max \left\{ \frac{\beta'}{\alpha_1}, \frac{\beta}{\alpha_2} \right\} \geq \max \left\{ \frac{\beta}{\alpha_1}, \frac{\beta'}{\alpha_2} \right\}$.
- 2) $\frac{\beta'}{\alpha_1} + \frac{\beta}{\alpha_2} \geq \frac{\beta}{\alpha_1} + \frac{\beta'}{\alpha_2}$.

A generalization of Lemma V.1 to k requested periods leads to the following key theorem:

Theorem V.1: For each frequency demand vector \mathcal{A} there exists an optimal tree T , an integer $2 \leq k \leq n$, and a node v such that the children of v in T are the clients with the smallest k shares.

The optimal tree algorithms rely on the result of Theorem V.1. The idea is to coalesce k clients with the smallest share demands into a new client, and then solve the new problem recursively. The only difference between algorithms for the different measures is the transformation of share demands.

In the algorithm for MAX, coalescing k clients with share demands (a_1, \dots, a_k) produces the new client with the share demand

$$a'_k = k \cdot \max \{a_1, \dots, a_k\}.$$

On the other hand, for AVE, coalescing k clients with share demands (a_1, \dots, a_k) produces the new client with the share demand

$$a'_k = \sqrt{k(a_1^2 + \dots + a_k^2)}.$$

Pseudo code for the optimal algorithm for the MAX measure is presented in Figure 11. Algorithm optAve for the AVE measure is identical, except for the computation of a'_k .

Algorithm optMax

input: $\mathcal{A} = (a_1, a_2, \dots, a_n)$

output: schedule tree T , and its MAX value val

if $n = 1$ return $(\{a_1\}, a_1)$

for $2 \leq k \leq n$

$M_k \leftarrow k$ smallest elements of \mathcal{A}

$a'_k \leftarrow k \max\{a \mid a \in M_k\}$

$\mathcal{A}_k \leftarrow \mathcal{A} \cup \{a'_k\} \setminus M_k$

$(T_k, val_k) \leftarrow \text{optMax}(\mathcal{A}_k)$

$k^* \leftarrow \arg \min \{val_k \mid 2 \leq k \leq n\}$

$T \leftarrow T_{k^*}$, where nodes corresponding to elements of M_{k^*}

are added as children of the node corresponding to a'_{k^*} in T_{k^*}

return (T, val_{k^*})

Fig. 11. An optimal algorithm for the MAX measure

We summarize the correctness and complexity of optMax in the following theorem. (A similar result holds for optAve; we omit details.)

Theorem V.2: Let \mathcal{A} be a frequency request vector of n clients, let T be the schedule tree generated for \mathcal{A} by optMax, and let $\mathcal{B}(T)$ be the granted frequency vector implied by T . Then for any schedule tree T' for n clients $\text{MAX}_{\mathcal{A}, \mathcal{B}(T)} \leq \text{MAX}_{\mathcal{A}, \mathcal{B}(T')}$. Furthermore, time complexity of optMax is $O(2^n)$.

B. Heuristics

We now describe various ways we used to reduce the running time of the optimal algorithms by restricting their exhaustive search. At each recursive step our algorithms coalesce k clients with the smallest share demands and continue recursively. The optimal algorithms try each $2 \leq k \leq n$, resulting in exponential running time. To save time, we test only some restricted subsets of values for k . We now list all our heuristics.

We use the following convention in naming algorithms: the suffixes Ave and Max denotes the measure targeted by the algorithm.

bin: The bin algorithm performs recursive calls for $k = 2$ only, i.e., it generates the best possible binary trees. The bin algorithm (which resembles the Huffman algorithm [13]) has $O(n \log n)$ time complexity, since there is no fork of choices in the recursive step. It is known [8] that some binary tree schedules have approximation ratio of at most $\frac{4}{3} + \frac{2}{3} \max\{a_i\}$ for the AVE measure. Since binAve produces the best possible binary tree schedule, it follows that its approximation ratio is never worse than $\frac{4}{3} + \frac{2}{3} \max\{a_i\}$. (Simulations show that practically, the performance of the algorithm is much better.)

rr: We found that the bin algorithms perform badly especially when all remaining clients have similar demands, and this gave rise to the following algorithm: at each step, the algorithm tries both the recursive call, and the round robin option (which will finalize the tree). We add the *rr*- prefix to names of these algorithms. Obviously, these algorithms outperform the corresponding algorithms without the round robin option, since they examine a larger set of trees. The asymptotic complexity remains the same.

binMixed: Observe that for a small number of clients the running time of the optimal algorithm is good enough. This

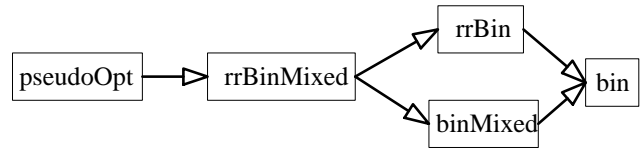


Fig. 12. The hierarchy among heuristic algorithms.

leads to the following algorithm: in the recursive step where ℓ clients are considered, recursively call the bin algorithm if $\ell > \log(n \log n)$, otherwise call the opt algorithm. Since the opt algorithm is called with a logarithmic number of clients, the running time of the resulting algorithm is $O(n \log n)$. binMixed algorithms achieved very good performance in our simulations.

pseudoOpt: The optimal algorithms choose at the recursive step the best k by calling optimal algorithms for reduced problems. The idea in pseudoOpt is to check, at each step, what is the best k according to some heuristic. Specifically, pseudoOpt tests the k smallest elements for all k , by coalescing them and applying rrBinMixed. The best k is chosen, the corresponding clients are coalesced, and the process continues. This heuristic algorithm clearly outperforms all other heuristics we presented; its running time is $O(n^3 \log n)$.

The hierarchy among the algorithms is described in Fig. 12. An arrow from one algorithm to another indicates that the first algorithm is superior in terms of results but inferior in terms of running time, as it considers a superset of the trees considered by the second.

C. Simulations

We study a performance of our heuristics for two distributions of share demands: the Zipf distribution [11], [21] and the uniform distribution, under both the MAX and AVE measures.

The Zipf distribution [11], [21]: This distribution depends on a *skew coefficient* θ . The value of the share of the i -th client is proportional to $i^{-\theta}$. That is,

$$a_i(\theta) = \frac{(1/i)^\theta}{\sum_{j=1}^n (1/j)^\theta}.$$

We use $\theta = 0.8$ in our simulations [11].

The uniform distribution: Each client chooses an absolute share demand a'_i , which is later normalized. Formally, $a'_i \sim U(0, 1)$ and $a_i = \frac{a'_i}{\sum_{j=1}^n a'_j}$. We repeat each experiment for the uniform distribution 10 times, and then average results.

In general, we found that the relative quality of the algorithms is the same for both the uniform and Zipf distributions, for both the MAX and AVE measures. Quantitatively, the algorithms get better results for AVE, and their behavior is more predictable under Zipf. We therefore give only a few representative graphs. Our empirical results (Figures 13 and 14) follow the hierarchy of heuristic algorithms (Figure 12) for both distributions considered. The weakest results are due to the bin algorithms (about 1.45 for MAX and 1.043 for the AVE). While rrBin algorithms perform only slightly better than bin,

binMixed algorithms provide a significant improvement. Recall that rrBinMixed achieves the minimum between rrBin and binMixed: this can be seen in that the graphs of rrBinMixed and binMixed coincide. The pseudoOpt heuristic improves on the results of rrBinMixed heuristic algorithms as expected. To conclude, for both distributions, our best heuristic scheme produces schedules which are about over 0.5% the optimum for the AVE measure and about over 15% for the MAX measure. All algorithms behave similarly for both Zipf and Uniform distributions of share demands.

1) *Performance for the Broadcast Disks problem:* The Broadcast Disks problem [1] can be viewed as a relaxation of the AVE measure, in the sense that the schedule needs not be periodic. In this case, a b_i value represents only the *average* frequency for client i . The best known results for the Broadcast Disks problem, from simulations point of view, are based on some variations of a greedy heuristic (e.g., [18], [6]). The idea in the greedy algorithms is to compute, every time slot, for each client i , what will be the “cost” of the schedule resulting if i is now scheduled, where cost is computed according to the AVE measure. The client that minimizes the cost increase is scheduled. In the next paragraph we explain how we compare its output with our heuristic algorithms.

The greedy algorithm, in general, produces non-cyclic schedules. Each slot, it scans all clients in order to choose a page to schedule, i.e. $\Omega(n)$ accesses. In addition, non-trivial computations need to be carried out. Therefore, in many cases one computes the schedule up to some point, and repeats it [9]. We use the same approach in order to be able to measure a performance of the greedy heuristic. We found that the best performance is achieved by greedy when the schedule length is about $100n$. Following [9], we chose the schedule length to be the best point in the interval $[100n, 101n]$. We assume that the demand probability of pages follows the Zipf distribution. It is known that the optimal schedule for the Broadcast Disks problem, if exists, is a perfect schedule where page i is granted a share $q_i = \frac{\sqrt{p_i}}{\sum_{i=1}^n \sqrt{p_i}}$. Hence, we build a broadcast schedule by calling our heuristic algorithms with a frequency request vector $\mathcal{A} = \langle q_1, \dots, q_n \rangle$. Then we compare a ratio between average waiting time achieved by perfect schedules to one achieved by the greedy algorithm. It can be seen in Figure 15, that our algorithms perform very well compared to greedy and even give better results for some database sizes.

VI. DISCUSSION AND OPEN PROBLEMS

In this paper we considered several aspects of a tree scheduling problem. First, we presented a constant amortized time algorithm for choosing the next client to schedule, given a schedule tree, using only polynomial space for most practical trees. Second, we considered a problem of constructing good schedule trees and describe an exponential time algorithm to find the optimal schedule tree. Then we proposed several polynomial-time heuristic algorithms based on the optimal algorithms. We tested our solutions by simulation. We concluded for the Zipf and Uniform frequency distributions, our approximation algorithms perform very well.

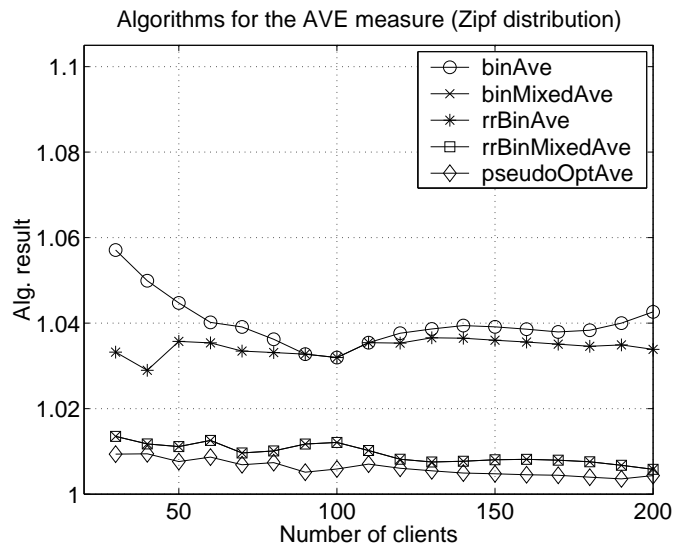


Fig. 13. The performance of algorithms for large number of clients for the AVE measure (Zipf distribution).

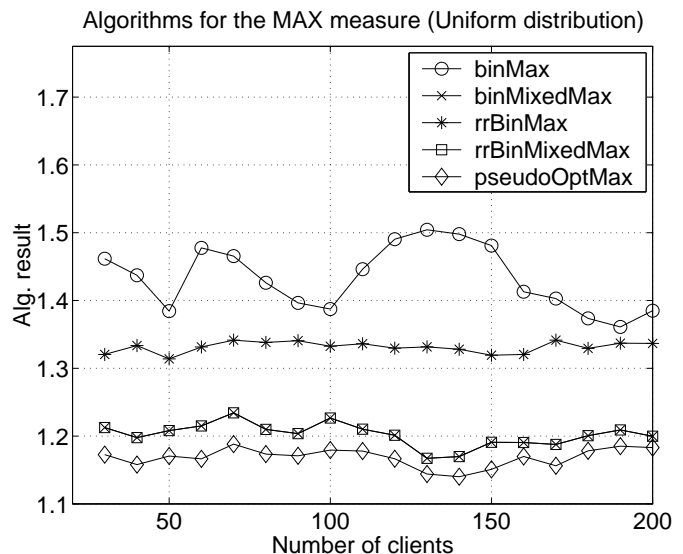


Fig. 14. The performance of algorithms for large number of clients or the MAX measure (Uniform distribution).

Open problems and further research:

- 1) We give a constant amortized time algorithm for constructing perfect schedules from trees and prove that it uses polynomial space for practical tree topologies. Is there a constant time polynomial space algorithm for all trees?
- 2) In our performance measures we “rewarded” the system for clients who received more than their requested share (by allowing $\rho_i < 1$). The measure of $\rho_i = \max \{1, \beta_i/\alpha_i\}$ (which means “free upgrades”) arises in many applications, and it seems worthwhile to study periodic scheduling for it.
- 3) It is known that deciding whether there exists a perfect schedule for given values of shares is NP-hard. Is it NP-

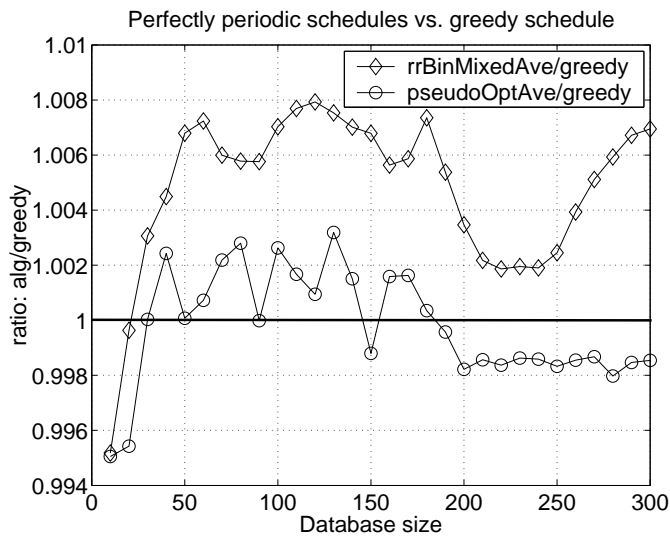


Fig. 15. The performance of our algorithms vs. greedy heuristic algorithm for Broadcast Disks problem. The heuristics are better than greedy whenever their graph is below 1.

hard to find the optimal tree?

- 4) There are perfect schedules that have no tree representation. Is there a better way to represent, design and use perfect schedules?

REFERENCES

- [1] S. Acharya, R. Alonso, M. J. Franklin, and S. Zdonik. *Broadcast Disks: data management for asymmetric communications Environment*. Proc. of the 1995 SIGMOD, 199-210, ACM Press, 1995.
- [2] S. Anily, C. A. Glass, and R. Hassin. *The scheduling of maintenance service*. Disc. Applied Math., Vol. 80, 27-42, 1998.
- [3] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [4] M. H. Ammar and J. W. Wong. *The Design of Teletext broadcast cycles*. Performance Evaluation 5(4), pp. 235-242, Dec. 1985.
- [5] M. H. Ammar and J. W. Wong. *On the optimality of cyclic transmission in Teletext systems*. IEEE Trans. on Comm., COM-35(1), 68-73, 1987.
- [6] A. Bar-Noy, R. Bhatia, J. Naor, and B. Schieber. *Minimizing Service and Operation Costs of Periodic Scheduling*. Proc. of the 9th ACM-SIAM Symp. on Disc. Alg. (SODA-98), 11-20, 1998.
- [7] A. Borodin and S. A. Cook. *A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation*. SIAM Journal on Computing 11(2), pp. 287-297, 1982.
- [8] A. Bar-Noy, A. Nisgav, and B. Patt-Shamir. *Nearly Optimal Perfectly-Periodic Schedules*. To appear in Proc. of the 20th ACM Symp. on Principles of Distr. Comp. (PODC 2001).
- [9] A. Bar-Noy, B. Patt-Shamir, and I. Ziper. *Broadcast Disks with Polynomial Cost Functions*. Proc. of the 19th INFOCOM, Vol. 2, 575-584, 2000.
- [10] *Bluetooth technical specifications, version 1.1*. Available from <http://www.bluetooth.com/>, Feb. 2001.
- [11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. *Web Caching and Zipf Distributions: Evidence and Implications*. Proc. of the 18th INFOCOM, Vol. 1, 126-134, 1999.
- [12] V. Dreizin. *Efficient Periodic Scheduling by Trees*. Master's Thesis, Tel-Aviv Univ., 2001, <http://www.eng.tau.ac.il/~vld/Th.html>
- [13] D. A. Huffman. *A Method for the Construction of Minimum Redundancy Codes*. Proc. of IRE, Vol. 40, 1098-1101, 1962.
- [14] C. Kenyon, N. Schabanel, and N. Young. *Polynomial-time approximation scheme for data broadcast*. Proc. of the 32th ACM Symp. on Theory of Computing (STOC-00), 659-666, 2000.
- [15] S. Khanna and S. Zhou. *On indexed data broadcast*. Proc. 30th ACM Symp. on Theory of Computing (STOC-98), 463-472, 1998.
- [16] C. J. Su and L. Tassiulas. *Broadcast Scheduling for Information Distribution*. Proc. of the 16th INFOCOM, Vol. 1, 109-117, 1997.
- [17] R. Tijdeman. *The Chairman assignment Problem*. Disc. Math., Vol. 32, 323-330, 1980.
- [18] N. Vaidya and S. Hameed. *Data broadcast: On-line and Off-line Algorithms*. Tech. Rep. 96-017, Dept. of Comp. Sc., Texas A&M Univ., 1996.
- [19] N. Vaidya and S. Hameed. *Log time algorithms for scheduling single and multiple channel data broadcast*. Proc. of the 3rd MOBICOM, 90-99, 1997.
- [20] W. Wei and C. Liu. *On a periodic maintenance problem*. Operations Research letters, Vol. 2, 90-93, 1983.
- [21] G. K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, Reading, MA, 1949.