Dispatching in Perfectly-Periodic Schedules

Zvika Brakerski, Vladimir Dreizin, Boaz Patt-Shamir

Department of Electrical Engineering, Tel Aviv University, 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, called the period of that client. Periodic schedules are useful in mobile communication where they can help save power in the mobile device, and they also enjoy the best possible smoothness. In this paper we study the question of *dispatching* in a perfectly periodic schedule, namely how to find the next item to schedule, assuming that the schedule is already given somehow. Simple dispatching algorithms suffer from either linear time complexity per slot or from exponential space requirement. We show that if the schedule is given in a natural tree representation, then there exists a way to get the best possible running time per slot for a given space parameter, or the best possible space (up to a polynomial) for a given time parameter. We show that in many practical cases, the running time is constant and the space complexity is polynomial.

1 Introduction

Consider a system with n clients and a single resource they share by means of time multiplexing. A schedule for such a system is called *perfectly periodic* (or just *perfect* for short), if each client i gets one time slot *exactly* every β_i time slots, for some β_i called the period of i. Perfect schedules are attractive for various reasons (see below). However, perfect schedules are, in general, non-trivial mathematical objects. It is known, for example, that even deciding whether a given set of periods admits a perfect schedule is NP-Hard [1]. The best known methods for finding perfect schedules produce schedules that are essentially hierarchical composition of round-robin schedules, called tree schedules [2]. A tree schedule is represented by a tree, where leaves correspond to clients, and the period of each client is the product of the degrees of the nodes on the path from the root to its leaf (see example in Figure 1 for some intuition; formal details are given in Section 2).

There are some algorithms to generate good schedule trees, according to certain target functions. In this paper, we study a different problem, called *dis-*

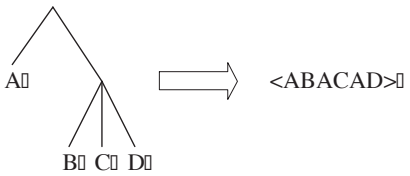

Fig. 1. *An example of a tree and its corresponding schedule. The clients are* $\{A, B, C, D\}$*, and the corresponding schedule has a cycle of length* 6*.*

patching, defined as follows. We assume that a perfect schedule is already given somehow. The task of the dispatching algorithm is to output, at each time slot, the identity of a client to be scheduled at that slot. To appreciate the problem, consider the following two naïve dispatching algorithms. One is to maintain, for each client, a variable that records when that client's next time slot is , and in each invocation of the dispatching algorithm, scan the clients one by one until we find the one whose turn has come up. The other extreme solution is to create a table that lists a complete cycle of the schedule, and use a pointer to run through it cyclically. The former solution maintains only $O(n)$ numbers each one having up to n bits (where n is the number of clients), but it may take $\Omega(n)$ time steps per each invocation. The latter solution, on the other hand, may require, in some cases, $2^{\Omega(n)}$ space, while taking only $O(1)$ time. Using a heap, it is not difficult to improve the space-efficient (first) solution to work in $O(\log n)$ time. However, in this paper we show that for tree schedules, one can enjoy the best of both worlds: roughly speaking, we present a scheme that can be given a time parameter and it finds the representation with the smallest space (up to a polynomial factor) that allows for a dispatching algorithm with the given time complexity; conversely, given a space parameter we can find the representation of the given size with the smallest time complexity of dispatching (up to a constant factor). Optimality here is with respect to our scheme, but in fact, we can show that in many cases that arise in practice, the time complexity is constant while the space complexity is polynomial. All that needs to be done is a pre-processing phase, that prepares the data structures for the dispatching algorithm. Before we elaborate on our results, let us give a brief overview of the background.

1.0.0.1 Why perfect schedules. Perfect schedules are attractive from a few viewpoints, all due to the fact that mathematically, they are very simple to describe: the schedule of a client is completely specified by two numbers (period and offset). This inherent simplicity gives rise to several pleasing consequences; for example:

- In some sense, perfect schedules are the smoothest schedules, or in other words, they give the best discrete-time approximation possible to a continuous-time allocation of the resources. For a more thorough discussion on the fairness issue, see, for example, the "chairperson assignment problem" [3].

- In the context of "push systems" such as broadcast disks [4], the server schedules items, and a client that wishes to access a certain item must wait until that item is scheduled. If the schedule is perfectly periodic, it is extremely easy for the client to compute when will be the next occurrence of its desired item, assuming the existence of a global clock. This allows the client to switch its receiver off temporarily. Such a power saving mode of operation is particularly important when the clients are tiny mobile devices with limited power supply (see, for example, the "Sniff mode" in the Bluetooth protocol [5]).
- Even without a global clock, periodic schedules are amenable to efficient indexing schemes [6]. The idea of these schemes is to interleave "index items" among the regular items in the schedule, so that even if a client does not know its schedule, it can learn it with very little effort (where effort is measured by power consumption, modeled by the duration of actively listening to the items scheduled).

1.0.0.2 Related work. Ammar and Wong [7,8], motivated by Teletext systems, show that the optimal schedule is cyclic, and give an approximation algorithm for periodic scheduling. They do not treat the dispatching problem. Hameed and Vaidya [9,10] propose using Weighted Fair Queuing to schedule broadcasts (which results in non-perfect schedules). They consider the dispatching problem; their algorithm takes $\Theta(\log n)$ steps per time slot on average. Khanna and Zhou [6] show how to use indexing with periodic scheduling to minimize busy waiting, and they also give an approximation algorithm for designing periodic schedules. In the work of Bar-Noy *et al.* [2], the general notion of perfect periodicity is introduced, as well as the tree methodology. The main results in [2] are algorithms for tree schedule design whose resulting schedules have guaranteed performance.

There is a large body of research about schedules that must satisfy a given set of *average* periods of clients, while compromising the *perfect* periodicity property. For example, Liu and Layland [11] call a schedule "periodic" if every client with period β is scheduled exactly once in each time window of the form $[(k-1)\beta, k\beta_i - 1]$ where k is an integer. Baruah et al. [12] introduced smoother schedules called "Pfair schedules," where it is required that in a prefix of t time units of the schedule, a client with resource share b is guaranteed to have either $\lceil t \cdot b \rceil$ or $\lfloor t \cdot b \rfloor$ occurrences. The results are typically for clients that may require more than one time slot per period, and systems with multiple resources. The best dispatching time known for Pfair schedules (called "per-slot time complexity" in [12]) is $\Theta(\log n)$ for a single resource. Additional papers with algorithms for non-perfect scheduling, motivated by Broadcast Disks and related problems, are [4,1,13–15]. The machine maintenance problem [16,17] and the chairperson assignment problem [3] are also closely related to periodic scheduling.

1.0.0.3 Our results. We study dispatching in the case of schedules for a single resource, where each client gets a single time slot in a period. To motivate the problem of dispatching, we first prove that in the worst case, the length of the cycle of a periodic schedule is exponential in the number of clients; it may also be exponential in the length of the longest period. Our main result is a dispatching algorithm based on transforming the tree representation of a perfect schedule into a *dag* representation by finding a cut in the tree and "flattening" the part above the cut. There are two alternative dag constructions. One is based on a time parameter t. The space requirement of the dispatching algorithm is at most the square of the space required by the best dag to dispatch using time t, and the running time of the dispatching algorithm is at most $2t$. The other alternative gets a space parameter S, and it finds a schedule dag whose dispatch time at most twice the optimal for dags with space S; the size of the dag in this case is at most S^2. We also show that in some practical cases, the dispatching time is constant and the storage space is polynomial in the number of clients.

1.0.0.4 Paper organization. The remainder of the paper is organized as follows. In Section 2 we define the notions of perfectly periodic schedules and tree schedules. In Section 3 we bound the length of cycles of perfect schedules. In Section 4 we describe and analyze our dispatching algorithm, based on the concept of schedule dags. In Section 5 we explain how to create optimal schedule dags for a given time or space parameter. We conclude in Section 6.

2 Definitions and preliminaries

2.1 Schedules

A *schedule* is an infinite sequence $S = s_0, s_1, \ldots$, where $s_j \in \{1, 2, \ldots, n\}$ for all j. If $s_j = i$ we say that *client i is scheduled at slot j*, or equivalently, *slot j is allocated to client i*. A schedule is called *cyclic* if it is an infinite concatenation of a finite sequence $C = \left\langle s_0, s_1, \ldots, s_{|C|-1} \right\rangle$. C is called a *cycle* of S. In this paper we are concerned exclusively with cyclic schedules, and we refer interchangeably to a schedule and to its cycle. A schedule is called *perfectly periodic*, or *perfect* for short, if slots allocated to each client are equally spaced, i.e., for each client i there exist integers $\beta_i \geq 1$ and $0 \leq o_i < \beta_i$, such that i is scheduled in slot j if and only if $j \equiv o_i \pmod{\beta_i}$. In this case β_i is called the *period* of i, and o_i is called the *offset* of 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 *share* of client i.

For a set of numbers $n_1, \ldots, n_k$, we denote by $\mathsf{lcm}(n_1, \ldots, n_k)$ their least common multiple. For later reference, we state the following immediate property of perfect schedules.

Lemma 1 *Let S be a perfect schedule with periods $\beta_1, \ldots, \beta_n$. Then the minimal cycle length of S is $\mathsf{lcm}(\beta_1, \ldots, \beta_n)$.*

PROOF. We first show minimality. Let C be a cycle of S. Since S is perfect, $|C|$ is divisible by each period β_i, for $i = 1, \ldots, n$. It follows from the definition of least common multiple that $|C| \geq \mathsf{lcm}(\beta_1, \ldots, \beta_n)$. Next, let C' be a the first $\mathsf{lcm}(\beta_1, \ldots, \beta_n)$ time slots in S. We claim that C' is a cycle of S, i.e., $S = S'$ for the schedule $S' = C'C'C' \ldots$. We show that this is true by showing that the schedule of each client is the same in S and in S'. Let i be any client. By definition, there exists a number o_i such that slot t is allocated to client i in S if and only if $t \equiv o_i \pmod{\beta_i}$. By construction, in the schedule S', slot t is allocated to client i if and only if $t = o_i + k_1\beta_i + k_2|C'|$, where $0 \leq k_1 < \frac{|C'|}{\beta_i}$ and $k_2 \geq 0$ are integers. Since $k_2|C'|$ is divisible by β_i because $|C'| = \mathsf{lcm}(\beta_1 \ldots, \beta_n)$, we have that $k_1\beta_i + k_2|C'| \equiv 0 \pmod{\beta_i}$, and therefore t is allocated to i in S' if and only if $t \equiv o_i \pmod{\beta_i}$, which completes the proof. □

2.2 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 in a rooted tree 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, denoted $u = \mathsf{par}(v)$. The *degree* of a node v in a rooted tree, denoted $\mathsf{deg}(v)$, is the number of its children. A *leaf* is a node with degree 0. An *ordered* tree is a rooted tree where the edges emanating from each non-leaf node are numbered $0, 1, \ldots, \mathsf{deg}(v) - 1$. The edge numbers induce a number for each non-root node among its siblings. The number assigned to a non-root node u among its siblings is denoted $\mathsf{rank}(u)$. Note that $0 \leq \mathsf{rank}(u) < \mathsf{deg}(\mathsf{par}(u))$.

A set of nodes V in a rooted tree T is a called a *cut of T* if there is exactly one node of V on every root-leaf path in T. 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 each level j, such that each node in level j is either a leaf or has degree d_j.

2.3 Tree schedules

An ordered tree, along with a bijection between the leaves and clients, corresponds to a perfect schedule as follows (see example in Figure 1). The period of the root r, denoted $\beta(r)$, is 1. The period of a non-root node v, denoted $\beta(v)$, is given by $\beta(v) = \beta(\mathsf{par}(v)) \cdot \mathsf{deg}(\mathsf{par}(v))$. The offset of the root, denoted $o(r)$, is 0. To define the offset of a non-root node v we use its edge order: $o(v) = o(\mathsf{par}(v)) + \mathsf{rank}(v)\beta(\mathsf{par}(v))$. (See [2] for a justification of this transformation; an example is given below.) Having defined a period and an offset for each node in the tree, the schedule of the tree is given by the correspondence between leaves in the tree and clients in the schedule.

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. Edges are ordered left-to-right, and hence the offset of A is 0, the offset of B is 1, the offset of C is 3 and the offset of D is 5.

We refer to a tree that represents a schedule as a *schedule tree*, and to a schedule that can be represented by a tree as a *tree schedule*. If T is a schedule tree, then $C(T)$ denotes the cycle of its corresponding schedule.

Note that without loss of generality, we may assume that no node in a schedule tree has degree 1.

3 Bounds on the cycles of 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.

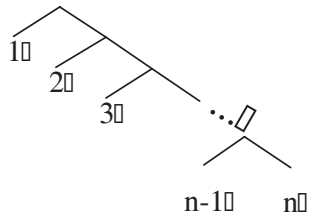

Fig. 2. *A tree whose cycle length is* 2^{n-1}.

We start by observing that there exist schedules with n clients whose cycle length is 2^{n-1}. Consider, for example, the schedule where the periods are defined as follows:

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

A tree corresponding to this schedule is depicted in Figure 2. The minimal cycle length for this schedule, by Lemma 1, is 2^{n-1}.

The next theorem shows that in some sense, the example of Figure 2 is the worst possible, including all "crazy" trees.

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

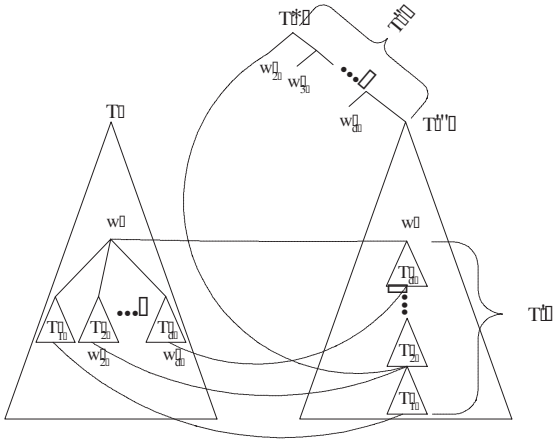

Fig. 3. *Construction for Theorem 2.*

PROOF. We prove that for any tree T with n leaves there exists a binary tree with n leaves whose cycle length is at least $|C(T)|$. This is sufficient, since the maximal cycle length for a binary tree with n leaves is 2^{n-1} (Figure 2 shows a worst case). So suppose that T has at least one non-binary node. We transform T into a tree T^* with n leaves, such that T^* has less non-binary nodes and $|C(T)| \leq |C(T^*)|$. Let $w \in T$ be a node with degree $d > 2$. Let $T_1, \ldots, T_d$ be the subtrees rooted at the children of w. Construct the new tree T^* as follows (see Figure 3).

(1) Let w_i be an arbitrary leaf of T_i, for $2 \leq i \leq d$.
(2) Construct a tree T' from $T_1, \ldots, T_d$ by replacing w_{i+1} in T_{i+1} with T_i, for all $1 \leq i < d$. This creates a "chain of trees" $T_d, T_{d-1}, \ldots, T_1$.
(3) Construct a binary tree T'' with d nodes and height $d-1$ (as in Figure 2). One of the two leaves in level $d-1$ is designated as "special", and all other leaves are $w_2, \ldots, w_d$.
(4) Construct a tree T''' from T by replacing $w \in T$ with T'.
(5) Construct T^* by replacing the special leaf in T'' with the root of T'''.

It is obvious from the construction that T^* has n leaves, and its number of non-binary nodes in T^* is strictly less than the number of non-binary nodes in T. We now analyze $|C(T^*)|$, the length of the cycle of the schedule corresponding

to T^*. For any tree G, $L(G)$ denotes the leaves of G, and $\beta_G(v)$ is the period in G of a leaf $v \in L(G)$. With these notations, we have:

$$\begin{aligned}
|C(T^*)| &= \mathsf{lcm}\,\{\beta_{T^*}(v) \mid v \in L(T^*)\} \\
&= 2^{d-1}\mathsf{lcm}\,\{\beta_{T'''}(v) \mid v \in L(T''')\} \\
&\geq 2^{d-1}\frac{1}{d}\mathsf{lcm}\,\{\beta_T(v) \mid v \in L(T)\} \\
&= 2^{d-1}\frac{1}{d}|C(T)| \\
&> |C(T)|
\end{aligned}$$

for $d > 2$. Therefore, T^* is a tree with n leaves and less non-binary nodes, but with greater corresponding schedule size. We can repeat this reduction step until all nodes have degree 2 or less. □

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 3 *There exists a tree T with largest period β, such that its corresponding schedule size is $2^{\Omega(\sqrt{\beta \ln \beta})}$.*

PROOF. Let m be such that $2m^2 \ln m = \beta$. This means that $m = \sqrt{\beta/2 \ln \beta} \cdot (1 + o(1))$. We construct a two-level tree as follows. The root has m children, and each child i of the root has q_i children, where the q_i's are chosen to be the m smallest primes larger $m \ln m$. Let $q \stackrel{\text{def}}{=} \max\{q_i\}$. By the Prime Number Theorem, we have that $q \approx 2m \ln m$, and hence the maximal period is $\beta = mq \approx 2m^2 \ln m$, as required. Now, since $q_1, \ldots, q_m$ are primes, their lcm is their product, and therefore, by Lemma 1, we have that the length of the schedule is at least $\prod_{i=1}^{m} q_i > (m \ln m)^m > 2^{\Omega(m \ln m)} = 2^{\Omega(\beta/m)}$. The result follows. □

4 Efficient dispatching of tree schedules

It is tempting to deny the difficulty inherent in dispatching: one natural algorithm is just to list a complete cycle of the schedule and go through it cyclically. However, the results of Section 3 show that this alternative is sometimes infeasible: exponential space is probably too much, even when the numbers of clients and the maximal period seem reasonable. [1] In this section we present a

[1] Consider the following schedule tree: the root has 12 children, whose degrees are the first twelve primes. While the corresponding schedule has less than 200 clients

space efficient dispatching algorithm and analyze its average running time; we then introduce the notion of *schedule dags* that allows us to save further on the average running time. Finally we show how to get the worst case running time to be as low as the average running time by some modest pre-processing.

4.1 The basic algorithm

Procedure dispatch
Input: A schedule tree T
Output: A client identifier
Code:
  $v \leftarrow \text{root}(T)$
  while v is not a leaf **do**
    $v \leftarrow$ moveToken(v)
  od
  return client corresponding to v

Procedure moveToken
Input: A non-leaf node $v \in T$
Persistent state: Token placement on edges of T
Output: A node in T
Side effect: Token placement altered
Invariant: For each non-leaf node, exactly one token on one of the outgoing edges
Code:
  Let $e_0, \ldots, e_{d-1}$ be the d outgoing edges of v
  Let $e_i = (v, u)$ be the outgoing edge of v with token
  Move token to edge $e_{(i+1) \bmod d}$
  return u

Fig. 4. *Procedures* dispatch *and* moveToken

We now describe a space-efficient algorithm for dispatching tree schedules. Pseudo code is given in Figure 4 (Procedure moveToken is separated because it is re-used in the final algorithm). The idea is to find the client to schedule by traversing the tree guided by *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 descends the tree starting from the root, by following the edges with tokens. In addition, each time an edge $e = (v, u)$ is crossed, the token is moved to the next outgoing edge of v, where "next" is interpreted cyclically using the the edge ordering. When a leaf is reached, it is output as the 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

and the maximal period less than 500, its cycle length is approximately $8 \cdot 10^{13}$.

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 correctness of dispatch in the following theorem.

Theorem 4 *Let T be a schedule tree and let c be the minimal cycle length of its corresponding schedule. Then starting with any valid initial token placement, c applications of procedure* dispatch *produce a cycle of S.*

PROOF. We show, by induction on the height of the tree, that the resulting schedule is perfectly periodic with respect to each client. For height 0, the claim is trivial. For the induction step, assume that the root has d children, and let $T_1, \ldots, T_d$ be subtrees rooted at the children of the root of T. Let i be any client, and suppose w.l.o.g. that i is a leaf in T_k. Consider first only the time slots in which dispatch visits T_k. In these time slots, by induction, the schedule of i is perfectly periodic with some period $\beta_k(i)$. By the code, dispatch visits T_k in a perfectly periodic fashion: the token associated with the root of T visits T_k exactly once every d time slots. It follows that the schedule of i in the full sequence of time slots is perfectly periodic with period $d \cdot \beta_k(i)$, and we are done. □

Clearly, the worst-case running time of dispatch is the height of T, and its space requirement is proportional to the size of the tree. For reasons that will become apparent in Section 4.3, we are interested in the *amortized* (average) time complexity of dispatch. For this, we need the following natural definition. Recall that b_i is the *share* of client i, defined by $b_i = 1/\beta_i$.

Definition 5 *Let T be a schedule tree with clients $1, \ldots, n$ whose shares are $b_1, \ldots, b_n$, respectively. Let $\ell(i)$ be the level of leaf i in T. The* tree entropy *of T, denoted $H(T)$, is*

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

For this notion of tree entropy, we have the following straightforward result.

Theorem 6 *Let T be a schedule tree. The total running time of $|C(T)|$ consecutive applications of* dispatch *is $O(|C(T)| \cdot H(T))$.*

PROOF. The running time of dispatch when it outputs a leaf i is proportional to the level of i. Since leaf i is visited exactly $b_i \cdot |C(T)|$ times in $|C(T)|$ consecutive applications of dispatch, the result follows. □

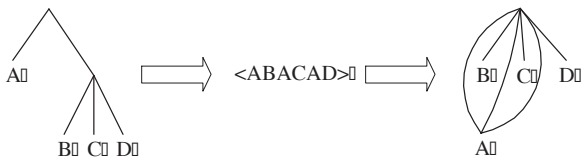

Fig. 5. *The schedule of Fig. 1 and its round-robin dag representation. Edges are oriented downwards, and their ranks are ordered left-to-right.*

The notion of tree entropy is a generalization of the information-theoretic concept of entropy: the (information-theoretic) entropy of a stochastic source of n symbols whose probabilities are $\{b_1, \ldots, b_n\}$ is defined by

$$H_2(T) = \sum_{i=1}^{n} b_i \log \frac{1}{b_i} \ .$$

(All logarithms in this paper are taken to base 2.) Obviously, if T is a binary tree, then $H(T) = H_2(T)$. In general, for any tree T whose non-leaf nodes have degree at least 2, we have that $H(T) \leq H_2(T)$, because $\ell(i) \leq \log \frac{1}{b_i}$ always. We can therefore conclude the following direct corollary.

Corollary 7 *The amortized running time of procedure* dispatch *on any schedule tree with n leaves is $O(\log(n))$.*

PROOF. Follows from the fact the the maximal entropy of a source with n symbols is at most $\log n$ (see e.g., [18]). □

4.1.0.5 An alternative algorithm for dispatching. Another implementation of dispatching for perfect schedules uses a heap containing all clients. Each client is inserted with its next slot number as its key: The initial "next slot" number is the offset of the client; when scheduled, the next slot number is simply the current time plus the period. To keep space complexity bounded, the numbers are reduced modulo the cycle length $|C|$ when their minimum is larger than $|C|$. This approach is inferior to algorithm dispatch from the bit-complexity point of view, since the numbers that the heap-based algorithm manipulates have $\Omega(\log |C|)$ bits, which in turn may be as high as $\Omega(n)$ (by Theorems 2 and 3). More importantly, the heap-based algorithm is not suited for the extensions we present next.

4.2 Schedule dags

We now 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. 5 for an example). Formally, a *rooted dag* is a directed acyclic graph with exactly one node without any incoming edge called *root*. An *ordered dag* is a rooted dag where the edges outgoing from each non-leaf node v are numbered $0, \ldots, \mathsf{deg}(v) - 1$ (recall that $\mathsf{deg}(v)$ is the number of edges outgoing from v). We note that there may be parallel edges in an ordered dag. A *schedule dag* is an ordered dag with a distinct client associated with each leaf. It is straightforward to verify that algorithm dispatch 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.

Our constructions, however, *do* guarantee that the schedule dags we generate correspond to perfect schedules. Intuitively, the idea is as follows. We take the schedule tree and cut it in a way that will be specified later. The tree above the cut is flattened by creating a full cycle list of its corresponding schedule. Then the cycle listing (in which each client may represent a subtree) is re-cast into a dag (see Figure 6). Finally, the dag is expanded back by "hanging" the subtrees previously cut back on their roots.

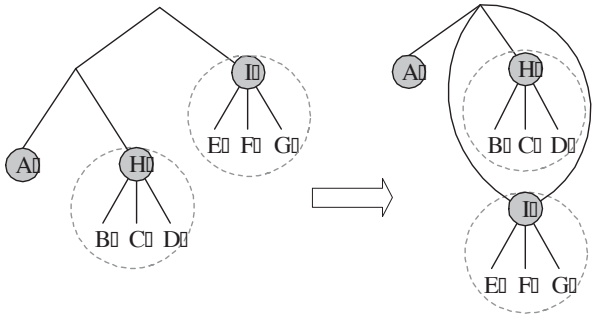

Fig. 6. *Example of* prepTree*, with the nodes in the cut shaded. All edges ore oriented downwards. Dashed circles denote compound leaves. Left: Tree schedule. Right: the resulting schedule dag.*

We call a set V of nodes of a tree T a *cut* if every root-leaf path contains exactly one node from V. In Algorithm prepTree, we use a function findCut that gets a tree T (and a time parameter t, which is not relevant at this stage), and returns a cut of T. In this section we do not specify the way findCut is implemented: this will be done in Section 5.

The main tools we use in the algorithm below are the functions contract, expand, and rrDag, defined as follows.

- The function $T' = \mathsf{contract}(T, V)$ receives a tree T and a cut V of T, and returns a new tree T' in which each subtree of T rooted at a node $v \in V$ is represented by a single *compound* leaf.
- The function $T = \mathsf{expand}(T', V)$ is the inverse of contract: 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 $T'' = \mathsf{rrDag}(T')$ converts a schedule tree T' into a *round robin dag* T'' as follows. The nodes of T'' comprise a single root r, and its leaves are exactly the leaves of T'. There are no other nodes in T''. The important things about T'' are its edges and their order: to define that, we compute the full cycle $C(T')$ of T' (this can be done, for example, by $|C(T')|$ applications of dispatch). We add $|C(T')|$ edges outgoing from r to the leaves. The order of the edges is defined by $C(T')$: the first edge is connected to the node corresponding to the first entry in $C(T')$ etc. (see Figure 5 for an example). The number of edges incoming into a a leaf in T'' is exactly the number of times its corresponding client appears in $C(T')$.

For formal details, see pseudo code in Figure 7.

Algorithm prepTree
Input: A schedule tree T, and time parameter t
Output: A schedule dag
Code:
  $V \leftarrow \mathsf{findCut}(T, t)$
  $T' \leftarrow \mathsf{contract}(T, V)$
  $T'' \leftarrow \mathsf{rrDag}(T')$
  return $\mathsf{expand}(T'', V)$

Fig. 7. *Preprocessing procedure converting a schedule tree to a schedule dag. Algorithm* findCut *is defined in Section 5.*

It is straightforward to verify the correctness of algorithm prepTree.

The time and space complexity of the dags generated by prepTree are stated in the following lemma.

Lemma 8 *Let T be a schedule tree and suppose that for some t, we have that $\mathsf{findCut}(T, t) = \{v_1, \ldots, v_k\}$. Let $T_1, \ldots, T_k$ be the subtrees rooted at $v_1, \ldots, v_k$, respectively. Then the amortized running time of* dispatch *on $\mathsf{prepTree}(T, t)$ is $\Theta(\sum_{i=1}^{k} \frac{1}{\beta(v_i)} H(T_i))$. The space complexity of $\mathsf{prepTree}(T, t)$ is $\Theta(n + \mathsf{lcm}\,\{\beta(v_i) \mid v_i \in V\})$.*

PROOF. The running time of dispatch is proportional to the depth of the leaf it outputs. With probability $1/\beta(v_i)$, this leaf is in T_i. The result follows by noting that the average depth of T_i is $H(T_i)$ by Theorem 6. The space complexity is proportional to the size of the dag generated by prepTree, which in turn is $O(|T''| + n)$, since expansion doesn't add any part of the tree more than once. The size of T'' is the cycle length of T', which is $\mathsf{lcm}\,\{\beta(v_i) \mid v_i \in V\}$ by Lemma 1. □

4.3 From amortized time to worst case time

In Section 4.1 we bounded the average running time of dispatch. Having a low average time seems to be useful only if the worst-case running time is still feasible. In this section we show that with $O(n)$ additional preprocessing, dispatch can be modified to run with worst-case time equal to the average time, which is the best possible.

The algorithm is as follows. Without loss of generality, let us define a unit time to be the worst-case time of an invocation of moveToken, i.e., the amount of time it takes for dispatch to visit a node in a schedule dag. Let $\bar{t}$ be the average running time of dispatch on the input schedule dag. The modified algorithm maintains a work-ahead FIFO buffer. In a pre-processing stage, the algorithm runs a sequence of invocations of dispatch for a pre-defined number of *total* time units: all client names that are output during these invocations are enqueued in the buffer, and the last state in which the algorithm paused is recorded. Then, in each scheduling time slot, Algorithm dispatch resumes its operation from the last recorded state, and runs for $\bar{t}$ time units. Again, this may comprise more than one invocation, and the last state reached is recorded. All outputs produced by dispatch during this operation are enqueued in the buffer. In addition, in each scheduling time slot, the algorithm dequeues a client identity from the head of the buffer and outputs it.

The only possible problem with the *correctness* of this approach is whether we get "underflows," i.e., situations in which the buffer is empty when it needs to make an output. Clearly, underflows can be avoided if the initial buffer is filled with the complete cycle—but this solution may be too expensive in terms of space, defeating our original goal. Note, in addition, that even if there are no underflows, it may be the case that during the operation of the algorithm, the work-ahead buffer gets very large. Thus, the other difficulty is *space complexity*: what is the size required by the work-ahead buffer to avoid overflows.

The following lemma shows that both these difficulties can be easily solved.

Lemma 9 *Let T be a schedule dag with n leaves. Let $\bar{t}$ be the average number of time units per time slot required by* dispatch *on T. Then for any $l \in \mathbb{N}$, l consecutive applications of* dispatch *on T required at most $l \cdot \bar{t} + 2n - 2$ time units and at least $l \cdot \bar{t} - (2n - 2)$ time units.*

PROOF. Let V be a set of all non-root nodes of T. Consider l consecutive applications of dispatch on T. For node $v \in V$ let $\beta(v)$ be a period that is associated with node v, and let n_v be a number of times that algorithm visits v during l invocations. To prove the lemma, we use the following two

observations: First, note that the time that elapses during l invocations of dispatch on T is $\sum_{v \in V} n_v$. And second, we note that for all $v \in V$, in any l consecutive invocations of dispatch, we have that

$$\left\lfloor \frac{l}{\beta(v)} \right\rfloor \leq n_v \leq \left\lceil \frac{l}{\beta(v)} \right\rceil \ .$$

Hence, the time that elapses in l invocations of dispatch on T is at most

$$\begin{aligned}
\sum_{v \in V} n_v &\leq \sum_{v \in V} \left\lceil \frac{l}{\beta(v)} \right\rceil \\
&< \sum_{v \in V} \left(\frac{l}{\beta(v)} + 1 \right) \\
&= l \cdot \sum_{v \in V} \frac{1}{\beta(v)} + |V| \\
&= l \cdot \bar{t} + |V| \ .
\end{aligned}$$

Similarly, $\sum_{v \in V} n_v > l \cdot \bar{t} - |V|$. The lemma follows from the fact that $|V|$, the number of non-root nodes in a schedule trees is at most twice the number of leaves. For schedule dags, we compare it to the original schedule tree: the number of leaves is the same in both, and the number of non-leaf nodes in a schedule dag is no more than the number of non-leaf nodes in the original tree. □

The consequence of Lemma 9 is twofold. First, the upper bound means that it suffices to run the dispatch algorithm for $2n-2$ steps in the pre-processing stage to fill the work ahead buffer to avoid underflows, so long as we let dispatch run $\overline{t}$ steps in each time slot. And secondly, the lower bound means that whatever is the current state of the work-ahead buffer, the number of items it contains will never increase by more than $2n - 2$. Pseudo code for the full algorithm is given in Figure 8, using the standard queue operations insertToTail that inserts a new element at the tail of the queue, and removeFromHead that removes the head element from the queue and returns it. To allow resumption of an invocation, the algorithm maintains its current state using the variable v. The only new detail in the implementation is the protection from overflows: since we run the algorithm for $\lceil \bar{t} \rceil$ steps per time slot, we may be working too much. To avoid that, we impose the restriction to stop inserting new items when the buffer already contains $2n$ client identifiers.

We summarize with the following statement.

Lemma 10 *Let T be an n-clients schedule dag produced by Algorithm* prepTree*, and suppose that its size is S and its average running time is $\bar{t}$. Then there*

Input: A schedule dag T with n leaves, whose amortized running time is $\bar{t}$
Output: A client at each time slot
Persistent state:
A FIFO buffer Q that can hold up to $2n$ client identifiers
a node pointer v

Code for Preprocessing:

```
c ← 0
v ← root(T)
while c < 2n do
    v ← moveToken(v)
    c ← c + 1
    if v is a leaf then
        insertToTail(Q, v)
        v ← root(T)
    fi
od
```

Code for Per-Slot Invocation:

```
c ← 0
while c < t̄ and |Q| < 2n do
    v ← moveToken(v)
    c ← c + 1
    if v is a leaf then
        insertToTail(Q, v)
        v ← root(T)
    fi
od
return removeFromHead(Q)
```

Fig. 8. *The full dispatching scheme. Code for* moveToken *is given in Figure 4.*

exists a dispatching algorithm for T with space $O(S)$, worst-case time $O(\bar{t})$, and preprocessing time $O(n)$.

In light of Lemma 10, let us refer to the size a schedule dag T as the *space complexity of T*, and to the amortized running time of dispatch on T as the *time complexity of T*.

5 Finding Good Cuts

In this section we show how to find cuts that will ensure dags with simultaneously low time and space complexities. We first describe a general solution which is guaranteed to be close to optimal, and then we point out a few important special cases where we can bound simultaneously the time complexity by a constant and the space complexity by a polynomial.

5.1 The bi-criteria optimization

The challenge in finding a good cut is to simultaneously reduce the average time and space complexities. To do that, we represent the problem of finding a good cut as a bi-criteria integer linear program. It turns out that a simple rounding of the relaxed linear program suffices.

The integer program is based on representing numbers by their prime factor-

ization: Let p_k denote the kth prime number. Then for a given number m, let $e_1(m), e_2(m), \ldots$ be the unique integers such that $m = \prod_k p_k^{e_k(m)}$. To represent a node i in the tree whose period is $\beta(i)$, we will use the numbers $e_k(\beta(i))$. Note that since the β values are already given as a product of node degrees, all we essentially have to do to get this representation is to factor the degrees, which can be done in time polynomial in n (since degrees in a tree are at most $n-1$).

We first present an integer program based on a given schedule tree T and a parameter t. Let n be the number of leaves in T, and let m be the number of nodes in T.

Variables: a variable x_i for each node i in the tree, and a variable y_k for each prime smaller than n. (In the solution, $x_i = 1$ if i is above or in the cut, and $x_i = 0$ otherwise; y_k is the exponent of p_k in the prime factorization of the lcm of the nodes in the cut.)

Goal: Minimize the following quantity (p_k denotes the kth prime).

$$\sum_{p_k < n} y_k \log p_k \tag{1}$$

$$t \geq \sum_{i=1}^{m} \frac{1}{\beta(i)} \cdot (1 - x_i) \tag{2}$$

$$y_k \geq x_i \cdot e_k(\beta(i)) \qquad \text{for all } 1 \leq i \leq m \text{ and all } k \text{ s.t. } p_k < n \tag{3}$$

$$x_i \in \{0, 1\} \qquad \text{for all } 1 \leq i \leq m \tag{4}$$

We using the following concept in the analysis of our cut-producing algorithms.

Constraints: Definition 11 *Let T be a rooted tree and let $\{x_i \mid i \text{ is a node in } T\}$ be an assignment of real numbers to nodes in T. The assignment is called* normal *if for all nodes i we have $x_i \leq x_{\mathsf{par}(i)}$ and $x_i = x_j$ for all siblings j of i.*

The connection between normal assignments and cuts is the following. Suppose we are given a tree T with a normal assignment $\{x_i\}$ on its nodes, and we are also given a real number θ. Then the set of nodes i with $x_i > \theta$ such that $x_j \leq \theta$ for all children j of i defines a cut of T. We first apply this idea to the integer program above.

Lemma 12 *Let T be a schedule tree, and let t be a parameter. Then a solution to Eqs. (1–4) above finds a cut which defines a dag (by* prepTree*) with time complexity $t+1$ and space complexity that is minimal among all cuts with running time t or less. Moreover, the expression in Eq. (1) is the logarithm of that space complexity.*

PROOF. Consider any solution to the program. We first claim that if $\beta(j)$ divides $\beta(i)$ and $x_i = 1$, then without loss of generality we may assume that $x_j = 1$. This follows from the fact that if $\beta(j)|\beta(i)$, then $e_k(\beta(j)) \leq e_k(\beta(i))$ for all k and therefore the constraints over x_j (Eq. (3)) are satisfied even if $x_j = 1$. As a result of this claim, we may assume, without loss of generality, that if $x_i = 1$, then $x_{\mathsf{par}(i)} = 1$ and $x_j = 1$ for all siblings j of i. This means that there exists an optimal solution in which the x_i values are normal in the sense of Def. 11 above. We can therefore define the cut to consist of all nodes i with $x_i = 1$ and such that all their children j (if exist) have $x_j = 0$.

Next, note that from Constraint (3) we have that the target function (1) is exactly the logarithm of the lcm of the periods of the nodes in the cut. Finally, we claim that Constraint (2) means that the running time of dispatch on the cut generated by this program is at most $t + 1$: this follows from the fact the for any subtree T_ℓ, $H(T_\ell) = \sum_{i \in T_\ell} \frac{1}{\beta(i)}$. The extra time unit is due to the fact that the root of each T_ℓ is one edge away from the root of T. The result follows. □

Procedure findCut
Input: A schedule tree T and a time parameter t
Output: A cut of T
Code:
// Part 1: find fractional solution
For each node i, compute its period $\beta(i)$ and the exponents of its prime factorization $e_k(\beta(i))$
Solve the linear program of Eqs. (1–3, 5) (abort if infeasible)
// Part 2: normalize solution
for all nodes i **do**
  $x_i \leftarrow \max\{x_j \mid j$ is a node in a subtree rooted by either i or any of its siblings$\}$
od
// Part 3: extract cut
$L \leftarrow \emptyset$
for all nodes i **do**
  if $x_i > \frac{1}{2}$ **and** $x_j \leq \frac{1}{2}$ for all children j of i (if exist)
    then $L \leftarrow L \cup \{i\}$
  fi
od
return L

Fig. 9. *Finding a cut using linear programming*

To get a polynomial-time algorithm, we replace the integrality constraint (Eq. (4)) with the following linear constraint:

$$0 \leq x_i \leq 1 \qquad \text{for all } 1 \leq i \leq m \tag{5}$$

We solve the resulting linear program. We then transform the solution into a normal form. Finally, to get a cut, the real numbers obtained for x_i are

rounded to the nearest integer (0 or 1). See Figure 9 for pseudo code. For this procedure, we have the following result.

Theorem 13 *The cut returned by* findCut(T, t)*, when used by* prepTree*, defines a schedule dag whose time complexity is at most* $2t$ *and space complexity at most* S^2*, where* S *is the minimal size for dags based on* T *with time complexity at most* t*.*

PROOF. The first part of the algorithm is just a relaxation of the integer program. The next step is normalizing the solution. Clearly, after Part 2 of the algorithm, the solution is indeed normal. Moreover, applying the same arguments used in Lemma 12, we see that the solution remains feasible, and that the value of the target function has not increased. Since the solution is normal, Part 3 of the algorithm produces a cut. The theorem follows from Lemma 12 and the following two additional observations regarding the rounding rule. First, note that since each x_i is at most doubled, the y_k's need only be at most doubled to keep the solution feasible, and thus the space complexity (of which Eq. (1) is the logarithm) is at most squared. And secondly, all the $(1 - x_i)$ values are also at most doubled, and hence the running time (in Eq. (2)) is at most doubled too. The relation to the optimal solution follows from Lemma 12 and the fact that the optimal fractional solution is at least as good as the optimal integer solution. □

It is straightforward to reverse the order of optimization by fixing the space, and minimizing the time: the linear program is as follows. Let S be a parameter.

$$\begin{aligned}
&\text{Minimize} && \sum_{i=1}^{m} \frac{1}{\beta(i)} \cdot (1 - x_i) \\
&\text{subject to:} && \\
&&& \log S \geq \sum_{p_k < n} y_k \log p_k \\
&&& y_k \geq x_i \cdot e_k(\beta(i)) \qquad \text{for all } 1 \leq i \leq m \text{ and all } k \text{ s.t. } p_k < n
\end{aligned}$$

Similarly to Theorem 13, it can be shown that using the above linear program in procedure findCut results in a schedule dag with space complexity S^2 and time complexity which is at most twice the best possible for dags with space complexity S.

While the algorithm is Section 5.1 finds the best cuts under the given conditions, it is not readily clear what are the time and space parameters of these cuts. In this section we prove, by explicit construction of cuts, that in many cases the average time is constant and the space complexity is polynomial.

Consider the simple alternative algorithm for finding cuts, given in Figure 10: Essentially, the idea is to add a node to the cut if its period is larger than n and its parent period is smaller than n. Note that the parameter t used in findCut is not used here.

Procedure findCut_alt
Input: A schedule tree T with n leaves
Output: A cut of T
Code:
  $L \leftarrow \emptyset$
  for all non-root nodes i **do**
    if ($\beta(i) \geq n$ **and** $\beta(\mathsf{par}(i)) < n$) **or** ($i$ is a leaf **and** $\beta(i) < n$) ($\beta(i)$ is the period of i)
      then $L \leftarrow L \cup \{i\}$
    fi
  od
  return L

Fig. 10. *An alternative implementation of* findCut

Let prepTree_alt denote the algorithm prepTree where the call to findCut is replaced by a call to findCut_alt. It is easy to analyze the average running time this algorithm yields for dispatch.

Lemma 14 *Let T be any schedule tree with n leaves. Then the average time complexity of* dispatch *when applied to the dag* prepTree_alt(T) *is constant.*

PROOF. By Lemma 8, the average running time is proportional to $\sum_{i=1}^{k} \frac{1}{\beta(v_i)} H(T_i)$, where $\{v_i \mid 1 \leq i \leq k\}$ is the set of nodes in the cut and T_i is the subtree rooted at v_i, for each i. Let n_i denote the number of leaves in T_i. Note that $\sum_i n_i \leq n$. Note further that by the code of findCut_alt, for all nodes v_i in the cut we have that either $\beta(v_i) \geq n$, or else $H(T_i) = 0$ (the latter happens when v_i is a leaf). Hence the time complexity of dispatch when applied to the dag generated by prepTree_alt is proportional to

$$\sum_{i=1}^{k} \frac{1}{\beta(v_i)} H(T_i) \leq \sum_{i=1}^{k} \frac{1}{n} H(T_i)$$

$$
\begin{aligned}
&\leq \sum_{i=1}^{k} \frac{1}{n} \cdot \log n_i \\
&\leq \frac{1}{n} \sum_{i=1}^{k} n_i \\
&\leq 1 \ .
\end{aligned}
$$

□

The space complexity of the dags generated by cuts computed by prepTree_alt is more complicated to analyze. We offer here a proof of polynomial space for few simple cases, which we believe to cover most practical applications.

The first case is trees with bounded degree.

Lemma 15 *Let T be a schedule tree with n leaves and maximal degree Δ for some constant Δ. Then the size of the dag created by* prepTree_alt *is $n^{O(\frac{\Delta}{\ln \Delta})}$.*

PROOF. Let V be a cut returned by findCut_alt(T) and let $\beta_1, \ldots, \beta_k$ be the periods of the nodes $v \in V$. Denote by P_Δ the set of prime numbers not larger than Δ. By Lemma 8 the size of the dag created by prepTree_alt is $O\left(n + \mathsf{lcm}(\beta_1, \ldots, \beta_k)\right)$. Since degrees are bounded by Δ, we get that

$$\mathsf{lcm}(\beta_1, \ldots, \beta_k) = \prod_{q \in P_\Delta} q^{\gamma_q} \ ,$$

for some integers $\gamma_q \geq 0$. Let $\beta = \max\{\beta_1, \ldots, \beta_k\}$. Since $q^{\gamma_q} \leq \beta$ for all $q \in P_\Delta$, it follows that

$$\mathsf{lcm}(\beta_1, \ldots, \beta_n) = \prod_{q \in P_\Delta} q^{\gamma_q} \leq \beta^{|P_\Delta|} \ ,$$

and hence, the size of the dag is $O\left(n + \beta^{|P_\Delta|}\right)$. Since $\beta \leq n\Delta$ by the choice of Algorithm findCut_alt, and since by the Prime Number Theorem $|P_\Delta| \approx \frac{\Delta}{\ln \Delta}$, we get that the size of the dag is $O\left(n + (n\Delta)^{O(\frac{\Delta}{\ln \Delta})}\right) = n^{O(\frac{\Delta}{\ln \Delta})}$. □

We remark that the bound in Lemma 15 is the best possible in the following sense: there exist trees with maximal degree Δ such that the lcm of their leaves is roughly $\Omega(n^{|P_\Delta|})$.

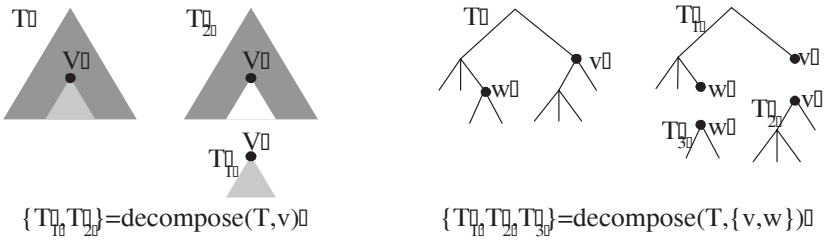

Fig. 11. *Example of* decompose *operation.*

The next case we consider is level-uniform trees. Recall that in a level-uniform tree, each level j is associated with a number d_j such that each non-leaf node in level j has exactly d_j children.

Lemma 16 *Let T be a level-uniform schedule tree. Then the size of maximal period in the dag created by* prepTree_alt *is $O(n^2)$.*

PROOF. First, note that the least common multiple of all periods of nodes in the cut returned by findCut_alt is exactly the *largest* of these periods: this follows immediately from the fact that the period of any node in level i divides the period of any node in level $i+1$. The lemma now follows from the fact that the maximal period of nodes in the cut is bounded by n (by the choice of findCut_alt) times the maximal degree in the tree, which is at most $n-1$. □

Finally, we consider various compositions of good cases. We use the following definition (see example in Figure 11).

Definition 17 *Let T be a schedule tree and let $v \in T$ be a node. Then* decompose(T, v) *is a pair T_1, T_2 of schedule trees where T_1 is the subtree of T rooted at v, and $T_2 = T - (T_1 - \{v\})$.*

We use a natural extension of decompose, which receives a set $V = \{v_1, \ldots, v_k\}$ of tree nodes, and applies decompose iteratively to get $|V|+1$ trees: first compute $(T_1, T_2) =$ decompose(T, v_1). Then, if $v_2 \in T_1$, we define decompose$(T, \{v_1, v_2\}) =$ $(T_2,$ decompose$(T_1, v_1))$.

With this definition, we can now give a sufficient condition on schedule trees for getting simultaneously constant running time and polynomial space. The proof entails a generalized version of prepTree.

Lemma 18 *Let T be a schedule tree with n leaves. Suppose that there exist a number k and a set of nodes V such that each root-leaf path in T contains at most k nodes from V. Let $\left\{T_1, \ldots, T_{|V|+1}\right\} =$* decompose$(T, V)$*, and suppose that each T_i has a dag with constant running time and space polynomial in n.*

Then there exists a schedule dag for T with size polynomial in n and running time $O(k)$.

PROOF. Consider the following preprocessing algorithm for T.

(1) $\left\{T_1, \ldots, T_{|V|+1}\right\} \leftarrow \mathsf{decompose}(T, V)$.
(2) For each i, let T_i' denote the schedule dag with polynomial space and constant running time, whose existence is guaranteed by the condition of the lemma.
(3) Return the dag that results from T after replacing each T_i with T_i'.

Obviously, the total space complexity $\sum_{i=1}^{k} |T_i'|$ is polynomial in n, since the size of each T_i' is polynomial by assumption, and since $k \leq n$. For the running time, consider an invocation of dispatch. At each time slot, dispatch descends the dag from the root towards leaves, passing at most k different schedule dags T_i'. Descending each T_i' takes constant time by assumption. The result follows. □

Corollary 19 *Let T be a schedule tree with n leaves. Suppose that there exist a number k and a set of nodes V such that each root-leaf path in T contains at most k nodes from V. Let $\left\{T_1, \ldots, T_{|V|+1}\right\} = \mathsf{decompose}(T, V)$, and suppose each T_i satisfies at least one of the following conditions.*

(1) T_i has a bounded degree, or
(2) T_i is level-uniform, or
(3) T_i has $O(\log n)$ leaves.

Then there exists a schedule dag for T with size polynomial in n and running time $O(k)$.

PROOF. By Lemma 18, it suffices to show that each T_i has a schedule dag with polynomial size and constant running time. Lemma 15 guarantees this if condition 1 holds, Lemma 16 guarantees this if condition 2 holds, and Theorem 2 guarantees this if condition 3 holds. □

We remark that most tree construction algorithms (e.g., [19,2]), produce trees that can be converted into linear-size, constant-time schedule dags, such as binary trees, trees with maximal degree 3, and trees that consist of a root whose children are roots of binary trees.

6 Conclusion

In this paper we considered the dispatching problem for perfectly periodic schedules. By studying properties of the cycles length we showed that a simple listing strategy may require exponential space. We gave a polynomial-space, entropy-time algorithm for dispatching tree schedules. To make dispatching even more efficient, we introduced the notion of schedule dags and showed how to derive them from schedule trees while optimizing either the space complexity for a given time complexity, or optimizing the time complexity for a given space complexity. We showed that in many practical cases, the time time complexity is constant and the space complexity is polynomial. We do not know whether this is always the case, or whether there exist some pathological examples where no constant-time, polynomial space dispatching is possible. In any case, we believe that this work helps validating the thesis that periodic scheduling in general, and tree scheduling in particular, can be implemented effectively and efficiently.

Acknowledgments

We thank Simon Litsyn for useful discussions and Jim Anderson for pointing to us the work on Pfair schedules.

References

[1] A. Bar-Noy, R. Bhatia, J. S. Naor, B. Schieber, Minimizing service and operation cost of periodic scheduling, in: Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 11–20.

[2] A. Bar-Noy, A. Nisgav, B. Patt-Shamir, Nearly optimal perfectly periodic schedules, Distributed Computing 15 (4) (2002) 207–220.

[3] R. Tijdeman, The chairman assignment problem, Discrete Mathematics 32 (1980) 323–330.

[4] S. Acharya, R. Alonso, M. Franklin, S. Zdonik, Broadcast disks: Data management for asymmetric communications environments, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, 1995, 1995, pp. 199–210.

[5] Bluetooth technical specifications, version 1.1., Available from http://www.bluetooth.com/ (Feb. 2001).

[6] S. Khanna, S. Zhou, On indexed data broadcast, in: Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98), New York, 1998, pp. 463–472.

[7] M. H. Ammar, J. W. Wong, The design of teletext broadcast cycles, Performance Evaluation 5 (4) (1985) 235–242.

[8] M. H. Ammar, J. W. Wong, On the optimality of cyclic transmission in teletext systems, IEEE Transaction on Communication COM-35 (1) (1987) 68–73.

[9] N. Vaidya, S. Hameed, Data broadcast: On-line and off-line algorithms., Technical Report 96-017, Department of Computer Science, Texas A&M University (1996).

[10] N. Vaidya, S. Hameed, Log time algorithms for scheduling single and multiple channel data broadcast, in: Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM'97), 1997, pp. 90–99.

[11] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM 20 (1) (1973) 46–61.

[12] S. K. Baruah, N. K. Cohen, C. G. Plaxton, D. A. Varvel, Proportionate progress: A notion of fairness in resourse allocation, Algorithmica 15 (1996) 600–625.

[13] A. Bar-Noy, B. Patt-Shamir, I. Ziper, Broadcast disks with polynomial cost functions, in: Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'00), Vol. 2, 2000, pp. 575–584.

[14] C. Kenyon, N. Schabanel, N. Young, Polynomial-time approximation scheme for data broadcast, in: Proceeding of the 32th Annual ACM Symposium on Theory of Computing (STOC-00), 2000, pp. 659–666.

[15] C. J. Su, L. Tassiulas, Broadcast scheduling for information distribution, in: Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'97), Vol. 1, 1997, pp. 109–117.

[16] W. Wei, C. Liu, On a periodic maintenance problem, Operations Research Letters 2 (1983) 90–93.

[17] S. Anily, C. A. Glass, R. Hassin, The scheduling of maintenance service, Discrete Apllied Mathematics 80 (1998) 27–42.

[18] T. M. Cover, J. A. Thomas, Elements of Information Theory., Wiley-Interscience, New-York, 1991.

[19] V. Dreizin, Efficient periodic scheduling by trees, Master's thesis, Department of Electrical Engineering, Tel Aviv University, available from http://www.eng.tau.ac.il/∼vld/Th.html (2001).