MISTA 2011

# Generating Better Cyclic Fair Sequences Faster with Aggregation

**Jeffrey W. Herrmann**

**Abstract** Fair sequences allocate capacity to competing demands in a variety of manufacturing and computer systems. This paper considers three problems that motivate generating cyclic fair sequences: (1) the response time variability problem, (2) routing jobs to parallel servers (the waiting time problem), and (3) finding balanced words. This paper discusses the similarities among these three problems and presents a general aggregation approach. Computational results show that using aggregation with stride scheduling both generates solutions that are more fair and reduces computational effort. The paper concludes with some ideas for applying the approach to related problems.

## 1 Introduction

In many applications, it is important to schedule the resource's activities in some fair manner so that each demand receives a share of the resource that is proportional to its demand relative to the competing demands. This problem arises in a variety of domains. A mixed-model, just-in-time assembly line produces different products at different rates. Maintenance and housekeeping staff need to perform preventive maintenance and housekeeping activities regularly, but some machines and areas need more attention than others. Vendors who manage their customers' inventory must deliver material to their customers at rates that match the different demands. Asynchronous transfer mode (ATM) networks must transmit audio and video files at rates that avoid poor performance.

The idea of fair sequences occurs in many different areas, including those mentioned above. Kubiak [1] reviewed the need for fair sequences in different domains and concluded that there are multiple definitions of a "fair sequence." This review discussed results for multiple related problems, including the product rate variation problem, generalized pinwheel scheduling, the hard real-time periodic scheduling problem, the periodic maintenance scheduling problem, stride scheduling, minimizing response time variability (RTV), and peer-to-peer fair scheduling. Bar-Noy et al. [2] discussed the generalized maintenance scheduling problem, in which the objective is to minimize the long-run average service and operating cost per time slot. Bar-Noy et al. [3] considered the problem of creating a schedule in which a single broadcast channel transmits each page perfectly periodically (that is, the interval between consecutive transmissions of a page is always the same). The goal is to find a feasible schedule so that the lengths of the scheduled periods are close to the ideal periods that will minimize average waiting time.

Jeffrey W. Herrmann
University of Maryland, College Park
E-mail: jwh2@umd.edu

In the periodic maintenance scheduling problem, the intervals between consecutive completions of the same task must equal a predetermined quantity. Wei and Liu [4] presented sufficient conditions for the existence of a feasible solution for a given number of servers. They also suggested that machines with the same maintenance interval could be replaced by a substitute machine with a smaller maintenance interval and that this replacement would facilitate finding a feasible solution. This concept, which was not developed into a solution algorithm, is similar to the aggregation proposed here. Park and Yun [5] and Campbell and Hardin [6] studied other problems that require the activities to be done strictly periodically and developed approaches to minimize the number of required resources. Park and Yun considered activities such as preventive maintenance tasks that take one time unit but can have different periods and different resource requirements. Campbell and Hardin studied the delivery of products to customers, where each delivery takes one day but the customers have different replenishment periods that must be satisfied exactly.

Waldspurger and Weihl [7] considered the problem of scheduling multithreaded computer systems. In such a system, there are multiple clients, and each client has a number of tickets. A client with twice as many tickets as another client should be allocated twice as many quanta (time slices) in any given time interval. Waldspurger and Weihl introduced the stride scheduling algorithm to solve this problem. They also presented a hierarchical stride scheduling approach that uses a balanced binary tree to group clients, uses stride scheduling to allocate quanta to the groups, and then, within each group, uses stride scheduling to allocate quanta to the clients. Although they note that grouping clients with the same number of tickets would be desirable, their approach does not exploit this. Indeed, the approach does not specify how to create the binary tree.

Kubiak [1] showed that the stride scheduling algorithm is the same as Jefferson's method of apportionment and is an instance of the more general parametric method of apportionment [8]. Thus, the stride scheduling algorithm can be parameterized.

The problem of mixed-model assembly lines has the same setup as the RTV problem but a different objective function. Miltenburg [9] considered objective functions that measure (in different ways) the total deviation of the actual cumulative production from the desired cumulative production and presented an exact algorithm and two heuristics for finding feasible solutions. Inman and Bulfin [10] considered the same setting but assigned due dates to each and every unit of the products. They then presented the results of a small computational study that compared earliest due date (EDD) schedules to those generated using Miltenburg's algorithms on a set of 100 instances. They show that the EDD schedules have nearly the same total deviation but can be generated much more quickly.

A more flexible approach in a cyclic situation is to minimize the variability in the time between consecutive allocations of the resource to the same demand (the same product or client, for instance). Corominas et al. [11] presented and analyzed the RTV objective function (which will be defined precisely below), showed that the RTV problem is NP-hard, and presented a dynamic program and a mathematical program for finding optimal solutions. Because those approaches required excessive computational effort, they conducted experiments to evaluate the performance of various heuristics. However, the Webster and Jefferson heuristics (described below) reportedly performed poorly across a range of problem instances. Garcia et al. [12] presented metaheuristic procedures for the problem. Kubiak [13] reviews these problems and others related to apportionment theory.

Unlike previous work on aggregation and cyclic fair sequences (reviewed in Section 2), this paper considers multiple measures of fairness and shows how using aggregation improves all of them.

## 2 Previous Work on Aggregation

Aggregation is a well-known and valuable technique for solving optimization problems, especially large-scale mathematical programming problems. Model aggregation replaces a large optimization problem with a smaller, auxiliary problem that is easier to solve [14]. The solution to the auxiliary model is then disaggregated to form a solution to the original problem. Model aggregation has been applied to a variety of production and distribution problems, including machine scheduling problems. In the context of cyclic fair sequences, aggregation combines products with equal demand. Although concepts similar to those of aggregation have been mentioned previously [4], the following papers are the first, in the context of cyclic fair sequences, to study the benefits of aggregation in a systematic way.

Herrmann [15] described the RTV problem and presented a heuristic that combined aggregation and parameterized stride scheduling. The aggregation approach combines products with the same demand into groups, creates a sequence for those groups, and then disaggregates the sequence into a sequence for each product.

For the single-server RTV problem, Herrmann [16] precisely defined the aggregation approach and described the results of extensive computational experiments using the aggregation approach in combination with the heuristics presented by [11]. The results showed that solutions generated using the aggregation approach can have dramatically lower RTV than solutions generated without using the aggregation approach. Herrmann [17] considered a more sophisticated "perfect aggregation" approach that can generate zero RTV sequences for some instances.

Herrmann [18] considered the RTV problem when multiple servers, working in parallel, are available, and presented a specific aggregation approach. The results showed that, in most cases, combining aggregation with other heuristics does dramatically reduce both RTV and computation time compared to using the heuristics without aggregation.

Herrmann [19] studied the problem of routing deterministic arriving jobs to parallel servers with deterministic service times, when the job arrival rate equals the total service capacity. This is known as the waiting time problem (WTP). This requires finding a periodic routing policy. The paper proposed an aggregation approach, and computational experiments showed that using aggregation not only reduces average waiting time but also reduces computational effort.

Herrmann [20] considered the balanced word problem (BWP), in which a set of letters must be assigned to positions in a sequence so that the letters occur at specified densities. Two different balance measures were used. The work used an aggregation approach in combination with different heuristics. The results of computational experiments showed that using the stride scheduling heuristic with aggregation generates the best solutions with the least computational effort (compared to the other heuristics).

## 3 Problem comparison

The RTV problem, WTP, and BWP have different objective functions but many similarities. All are concerned with finding a fair sequence. In a perfectly fair sequence, the positions assigned to one product (in the RTV problem), one server (in the WTP), or one letter (in the BWP) would be spaced perfectly evenly. The RTV would equal zero, jobs would never wait, and all of the letters would be balanced. But the discrete nature of the positions and the need to allocate positions to multiple products (servers, letters) may make this impossible.

The aggregation schemes presented in [16, 19, 20] are essentially the same. (The aggregation scheme in [18] adds an additional constraint to accommodate the multiple servers.) The similarity depends upon the similarity of the underlying problems, which becomes clear once we establish a "translation" between the terms used to describe the components of the problems, as shown in Table 1.

Table 1. Comparison of terminology for balanced word, RTV, and job routing problems.

| Problem component | Balanced word problem | RTV problem | Waiting time problem |
|---|---|---|---|
| Object being assigned to positions in the sequence | Letter | Product | Server |
| Frequency that must be satisfied | Density | Demand | Service rate |
| Position in the sequence | A position in a word | A time slot for processing by the server | An arriving job |

## 4 Problem Formulation

Given the translation above, we can formulate a general cyclic fair sequencing problem. We are given a set of *n* objects, each with a rational frequency $p_i$. The sum of the frequencies equals 1. Then, there exists a positive integer *T* and positive integers $x_1$, …, $x_n$ such that $p_i = x_i / T$ for *i* = 1,…, *n* and $\gcd(x_1, \ldots, x_n)$ = 1. Thus, $x_1 + \cdots + x_n = T$. Hereafter, we will describe an instance by the values of $(x_1, \ldots, x_n)$, with $x_1 \geq x_2 \geq \cdots \geq x_n$. We call each value the "count" of the corresponding object.

Given an instance $(x_1, \ldots, x_n)$, a sequence *s* of length *T* is a feasible sequence if exactly one object is assigned to each and every one of the *T* positions and each and every object *i* occurs exactly $x_i$ times in *s*.

The sequence will be repeated to make an infinite sequence. We seek a sequence that is as "fair" as possible. Different measures of fairness have been proposed, motivated by different applications.

The first two measures are related to the balanced word problem (BWP). The count balance measure considers how often a particular object appears in a subsequence. A sequence is *c*-balanced if, for any two subsequences *x* and *y* of the same size (number of positions), then $\left||x|_i - |y|_i\right| \leq c$, where $|x|_i$ is the number of times that *i* occurs in the factor *x* [13]. We will define the *count balance* of a sequence as the minimal such value of *c*. For example, the count balance of $(1231211321)^\infty$ equals 2 because $|11|_1 - |23|_1 = 2$ and $\left||x|_i - |y|_i\right| \leq 2$ for all subsequences *x* and *y* and all objects *i*.

The gap balance considers the gaps between consecutive occurrences of an object. Given a sequence, an object *a* is *m-balanced* if, whenever there exists an a-chain *aWa* in the sequence, any subsequence *W*' such that $|W'| = |W| + m + 1$ satisfies $|W'|_a \geq |W|_a + 1$. The sequence is *m*-balanced if each object is *m*-balanced [21]. We will define the *gap balance* of a sequence as the minimal such value of *m*. For example, in $(313132)^\infty$, the gap balance of the objects 2 and 3 equals 0, and the gap balance of the object 1 equals 2. Note that the subsequence 3 in the 1-chain 131 is 2 positions shorter than 323, the longest subsequence with no instance of the object 1. Therefore, the gap balance of this word equals 2.

Thus, we can describe BWP-count (and BWP-gap) as follows: Given an instance $(x_1, \ldots, x_n)$, find a finite word *S* of length *T* that minimizes the count balance (gap balance) of the infinite word *U* that is the infinite repetition of *S* subject to the constraints that exactly one object is assigned to each position of *S* and object *i* occurs exactly $x_i$ times in *S* for *i* = 1, …, *n*.

The complexity of BWP-count appears to be open. Given an instance, finding a word with a count balance that equals 1 requires finding a regular word. The complexity of this problem is open [13]. Likewise, the complexity of BWP-gap appears to be open. Given an instance, finding a word with an gap balance that equals 0 requires finding a constant gap word for $(x_1,\ldots,x_n)$. The complexity of the constant gap problem is open [1]. Nevertheless, these problems are related to the Periodic Maintenance Scheduling Problem, which is NP-complete in the strong sense [13], and the RTV problem, which is NP-hard [11].

The RTV of a feasible sequence equals the sum of the RTV for each object. If object *i* occurs at positions $\{p_{i1},\ldots,p_{id_i}\}$, its RTV is a function of the intervals between each position, which are $\{\Delta_{i1},\ldots,\Delta_{id_i}\}$, where the intervals are measured as $\Delta_{ik} = p_{ik} - p_{i,k-1}$ (with $p_{i0} = p_{id_i} - T$). The average interval for object *i* is $T / x_i$, so we calculate RTV as follows:

$$RTV = \sum_{i=1}^{n}\sum_{k=1}^{x_i}\left(\Delta_{ik} - \frac{T}{x_i}\right)^2$$

In the waiting time problem (WTP), there is a queueing system that consists of arriving jobs and a set of *n* parallel servers, each with its own queue. Starting at *t* = 0, one job arrives every time unit. Every arriving job must be routed to one of the servers at the moment that it arrives. If the server is idle, then the job immediately begins processing. If the server is busy, then the job waits in that server's queue. Server *i* has a service rate of $p_i$ jobs per time unit, so the job processing time is a fixed $1/p_i$ time units. The sequence specifies the servers to which the jobs should be routed.

The objective is to minimize the long-run average waiting time *W*. There exist periodic policies that minimize *W*, and *T* is the smallest period of such optimal periodic policies [22].

To simplify the evaluation of a policy, define $u_i^s$ as the total amount of time units that server *i* has been idle between time *t* = 0 and *t* = *s*. This depends upon how many jobs were routed to server *i*, when they arrived, and the server's processing time. Then, $S^t$, the total unused work capacity at time *t*, and *S*, the total unused work capacity, can be determined as follows:

$$S^t = \sum_{i=1}^{n} a_i u_i^t$$

$$d_i = \lim_{\tau\to\infty} a_i u_i^t$$

$$S = \lim_{t\to\infty} S^t = \sum_{i=1}^{n} d_i$$

When a periodic policy is applied, $W = S^t - (n-1)/2$ for $t \ge T-1$ (van der Laan, 2005).

The complexity of WTP also appears to be open. Given an instance of WTP, the question of finding a periodic policy with *W* = 0 requires finding a constant gap word for $(x_1,\ldots,x_n)$.

Consider, as an example, the following three-object instance: $(x_1, x_2, x_3)$ = (4, 3, 2) and *T* = 9. Consider the sequence $(112231123)^\infty$. The count balance of this sequence equals 2, and the gap balance equals 3 (because the gap balance of the object 1 equals 3). The RTV of this sequence equals 13.25. The average waiting time *W* = 7/9.

Now, consider the sequence $(121312123)^\infty$. The count balance of this sequence also equals 2 (because $|212|_2 - |131|_2 = 2$), but the gap balance equals 2 (because the gap balance of the object 2 equals 2). The RTV of this sequence equals 3.25. The average waiting time *W* = 4/9. Clearly, this sequence is more fair than the first one.

## 5 Aggregation

The aggregation approach used here repeatedly aggregates a set of objects until it cannot be aggregated any more. Each aggregation combines objects that have the same count into a group. These objects are removed, and the group becomes a new object in the new aggregated alphabet.

The key distinctions between the hierarchical stride scheduling algorithm of [7] and the aggregation approach presented here are (1) the hierarchical stride scheduling algorithm requires using the stride scheduling algorithm to disaggregate each group, since the objects in a group may have unequal counts and (2) the placement of objects in the tree is not specified. Because our aggregation scheme groups objects with equal counts, the disaggregation is much simpler. The limitation, however, is that the instance must have some equal count objects.

Also, unlike [3], our aggregation approach does not adjust the demands so that they fit into a single weighted tree (which would lead to a perfectly periodic schedule). Instead, the demands are fixed, and the aggregation creates a forest of weighted trees, which are used to disaggregate a sequence.

The aggregation procedure generates a sequence of instances $I_0, \ldots, I_H$. ($H$ is the index of the last aggregation created.) The aggregation can be done at most $n-1$ times because the number of objects decreases by at least one each time an aggregation occurs. Thus $H \le n-1$. Aggregation runs in O( $n^2$ ) time because each aggregation requires O($n$) time and there are at most $n-1$ aggregations.

The notation used in the algorithm that follows enables us to keep track of the aggregations in order to describe the disaggregation of a sequence precisely. Let $I_0$ be the original instance and $I_k$ be the $k$-th instance generated from $I_0$. Let $n_k$ be the number of objects in instance $I_k$. Let $B_j$ be the set of objects that form the new object $j$, and let $B_j(i)$ be the $i$-th object in that set. As the aggregation algorithm is presented, we describe its operation on the following five-object example: $I_0$ = (3, 2, 2, 1, 1), $n$ = 5, and $T$ = 9.

**Aggregation.** Given: an instance $I_0$ with counts $(x_1, x_2, \ldots, x_n)$.

1. Initialization. Let $k$ = 0 and $n_0 = n$.
2. Stopping rule. If all of the objects in $I_k$ have different counts, return $I_k$ and $H = k$ because no further aggregation is possible. Otherwise, let $G$ be the set of objects with the same count such that any smaller count is unique.

Example. With $k$ = 0, $G$ = {4, 5} because $x_4 = x_5$.

3. Aggregation. Let $m$ = |$G$| and let $i$ be one of the objects in $G$. Create a new object $n + k + 1$ with count $x_{n+k+1} = mx_i$. Create the new instance $I_{k+1}$ by removing from $I_k$ all $m$ objects in $G$ and adding object $n + k + 1$. Set $B_{n+k+1} = G$. $n_k = n_{k-1} - m + 1$. Increase $k$ by 1 and go to Step 2.

Example. With $k$ = 0 and $G$ = {4, 5}, the new object 6 has count $x_6 = 2 \times 1$=2. $B_6 = \{4,5\}$. The objects in $I_1$ are {1, 2, 3, 6}. When $k$ = 1, $G$ = {2, 3, 6}. The new object 7 has count $x_7 = 3 \times 2$ =6, and $B_7 = \{2,3,6\}$. The objects in $I_2$ are {1, 7}, which have different counts. Table 2 describes the instances created for this example.

Table 2. The counts for the five original objects in the example instance $I_0$ and the two new objects in the aggregate instances $I_1$ and $I_2$.

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ |
|---|---|---|---|---|---|---|---|
| $I_0$ | 3 | 2 | 2 | 1 | 1 | | |
| $I_1$ | 3 | 2 | 2 | | | 2 | |
| $I_2$ | 3 | | | | | | 6 |

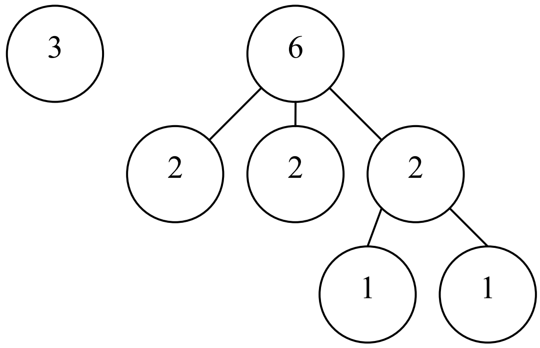


Figure 1. The forest corresponding to the aggregation of the example. The five leaf nodes correspond to the original objects in the example. The two parent nodes correspond to the new objects created during the aggregation. The two root nodes correspond to the objects remaining in the most aggregated instance.

At any point during the aggregation, the sum of the counts in a new instance will equal *T* because the count of the new object equals the sum of the counts of the objects that were combined to form it. We can represent the aggregation as a forest of weighted trees. There is one tree for each object in the aggregated instance $I_H$. The weight of the root of each tree is the sum of the counts of the objects in $I_0$ that were aggregated to form the corresponding object in $I_H$. The weight of any node besides the root node is the weight of its parent divided by the number of children of the parent. The leaves of a tree correspond to the objects in $I_0$ that were aggregated to form the corresponding object in $I_H$, and each one's weight equals the count of that object. The forest has one parent node for each new object formed during the aggregation, and the total number of nodes in the forest equals $n + H < 2n$. Figure 1 shows the forest corresponding to the aggregation of the (3, 2, 2, 1, 1) instance.

## 6 Disaggregation

When aggregation is complete, we must find a feasible periodic sequence for the aggregated instance $I_H$ and then disaggregate that sequence. This section presents the disaggregation procedure.

Let $S_H$ be a feasible sequence for the instance $I_H$. In particular, $S_H$ is a sequence of length *T*. Each position in $S_H$ is a object in the instance $I_H$. Disaggregating $S_H$ requires *H* steps that correspond to the aggregations that generated the instances $I_1$ to $I_H$, but they will, naturally, be considered in reverse order. We disaggregate $S_H$ to generate $S_{H-1}$ and then continue to disaggregate each sequence in turn to generate $S_{H-2}$, …, $S_0$. $S_0$ is a feasible sequence for $I_0$, the original instance.

The basic idea of disaggregating a sequence $S_k$ is to replace each new object with the objects used to form it. Object $n+k$ was formed to create instance $I_k$ from the objects in $B_{n+k}$, which were in $I_{k-1}$. It has $x_{n+k}$ positions in $S_k$. According to the aggregation scheme, $x_{n+k} = mx_i$, where $m = |B_{n+k}|$ and $i$ is one of the objects in $B_{n+k}$. The first position in $S_k$ assigned to object $n+k$ will, in the new sequence $S_{k-1}$, go to the first object in $B_{n+k}$, the second position assigned to object $n+k$ will go to the second object in $B_{n+k}$, and so forth. This will continue until all $x_{n+k}$ positions have been assigned. Each object in $B_{n+k}$ will get $x_{n+k}/m$ positions in $S_{k-1}$.

In the following algorithm, $j = S_k(a)$ means that object $j$ is in position $a$ in sequence $S_k$, and $B_{n+k}(i)$ is the $i$-th object in $B_{n+k}$.

**Disaggregation.** Given: The instances $I_0$, …, $I_H$ and the sequence $S_H$, a feasible sequence for the instance $I_H$.

1. Initialization. Let $k = H$.
2. Set $m = |B_{n+k}|$ and $i = 1$.
3. For $a = 0, \ldots, T-1$, perform the following step:
    a. If $S_k(a) < n+k$, assign $S_{k-1}(a) = S_k(a)$. Otherwise, assign $S_{k-1}(a) = B_{n+k}(i)$, increase $i$ by 1, and, if $i > m$, set $i = 1$.
4. Decrease $k$ by 1. If $k > 0$, go to Step 2. Otherwise, stop and return $S_0$.

Example. Consider the aggregation of the instance (3, 2, 2, 1, 1) presented earlier and the sequence $S_2$ = $(771771771)^\infty$, which is a feasible sequence for the aggregated instance $I_2$. When $k = 2$, $n+k = 7$, and $B_7 = \{2,3,6\}$. The positions in $S_2$ that are assigned to server 7 will be reassigned to servers 2, 3, and 6. The resulting sequence $S_1$ = $(231621361)^\infty$.

When $k = 1$, $n+k = 6$, and $B_6 = \{4,5\}$. The positions in $S_1$ that are assigned to object 6 will be reassigned to objects 4 and 5. The resulting sequence $S_0$ = $(231421351)^\infty$. Table 3 lists these three sequences.

As noted earlier, there are at most $n-1$ aggregations. Because each sequence disaggregation requires O($T$) effort, disaggregation runs in O($nT$) time in total.

Table 3. The disaggregation of sequence $S_2$ for instance $I_2$ in the example. The first row is $S_2$, a feasible sequence for instance $I_2$. The second row is $S_1$, a feasible sequence for instance $I_1$. The third row is $S_0$, a feasible sequence for instance $I_0$.

| $a$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| $S_2(a)$ | 7 | 7 | 1 | 7 | 7 | 1 | 7 | 7 | 1 |
| $S_1(a)$ | 2 | 3 | 1 | 6 | 2 | 1 | 3 | 6 | 1 |
| $S_0(a)$ | 2 | 3 | 1 | 4 | 2 | 1 | 3 | 5 | 1 |

## 7 Stride Scheduling

Studies of the RTV problem, WTP, and BWP have proposed various heuristics because finding optimal solutions is computationally expensive. The stride scheduling algorithm [7] has been used repeatedly and will be discussed here.

The parameterized stride scheduling algorithm builds a fair sequence and performs well at minimizing the maximum absolute deviation [1]. The algorithm has a single parameter δ that can range from 0 to 1. This parameter affects the relative priority of low-demand products and their absolute position within the sequence. When δ is near 0, low-demand products will be positioned earlier in the sequence. When δ is near 1, low-demand products will be positioned later in the sequence.

**Parameterized stride scheduling.** Given: an instance $(x_1, \ldots, x_n)$ and the parameter δ.

1. Initialization. $m_{i0} = 0$ for $i$ = 1, …, $n$. ($m_{ik}$ is the number of times that product $i$ occurs in the first $k$ positions of the sequence.)
2. For $k$ = 0, …, $T$-1, perform the following steps:
   a. Assign position $k+1$ to the lowest-numbered object $i*$ such that

   $$i^* = \arg\max_i \left\{ \frac{x_i}{m_{ik} + \delta} \right\}$$

   b. Set $m_{i*,k+1} = m_{i*,k} + 1$. For all $j$ not equal to $i*$, set $m_{j,k+1} = m_{jk}$.

The computational effort of the algorithm is O($nT$). For the five-object instance introduced earlier, the parameterized stride scheduling algorithm (with δ = 0.5) generates the sequence $(123145231)^{\infty}$.

Two versions of the parameterized stride scheduling algorithm are the well-known "Webster" and "Jefferson" apportionment methods [11]. These names refer to the traditional parametric methods of apportionment to which they are equivalent [1, 8]. The Webster heuristic uses δ = 0.5. The Jefferson heuristic uses δ = 1.

## 8 The Benefits of Aggregation

Experiments with thousands of instances of the BWP, RTV problem, and WTP have shown that using aggregation in combination with good heuristics both generates better sequences and reduces computational effort. In particular, using aggregation with parameterized stride scheduling algorithm generates good results.

The comparison of stride scheduling with other problem-specific heuristics was reported in [16, 18, 19, 20]. This paper discusses how aggregation with stride scheduling performs across a range of metrics of fairness. In addition, we will not consider the multiple-server RTV problem because the WTP and BWP have only a single sequence.

To generate the instances used in the computational study, we used the following scheme. First, we set the value of $T$ and the number of objects $n$. Then, for this combination, we generated $T-n$ random numbers from a discrete uniform distribution over $\{1, \ldots, n\}$. We then let $x_i$ equal one plus the number of copies of $i$ in the set of $T-n$ random numbers (this avoided the possibility that any $x_i = 0$). We generated 100 instances for each of 18 combinations of $T$ and $n$ shown in Table 4. All of the generated instances can be aggregated. (Note, in [16, 18, and 19], more instances were generated and used for testing aggregation.)

Table 4. Combinations of $T$ and $n$ used to generate instances.

| $T$ | $n$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 100 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
| 500 | 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 |

Table 5. Average number of aggregations for the instances in each problem set.

| $T$ | $n$ | Average number of aggregations |
|---|---|---|
| 100 | 10 | 2.66 |
| | 20 | 6.00 |
| | 30 | 5.71 |
| | 40 | 5.11 |
| | 50 | 4.03 |
| | 60 | 3.68 |
| | 70 | 3.23 |
| | 80 | 2.64 |
| | 90 | 2.07 |
| 500 | 50 | 10.77 |
| | 100 | 9.20 |
| | 150 | 7.39 |
| | 200 | 6.09 |
| | 250 | 5.10 |
| | 300 | 4.34 |
| | 350 | 3.84 |
| | 400 | 3.20 |
| | 450 | 2.69 |

For each instance, we constructed solutions as follows. First, we applied the stride scheduling algorithm (with $\delta = 0.5$ and with $\delta = 1$) to the instance (we call this the H solution). Next, we aggregated the instance. For the aggregate instance, we applied the heuristic to construct an aggregated solution. We disaggregated this solution to construct the AHD solution.

When considering the RTV measure, we also applied the exchange heuristic to the heuristic solution to construct the HE sequence. Finally, we applied the exchange heuristic to the AHD sequence to construct the AHDE sequence. The exchange heuristic significantly reduces the RTV of the solutions generated by the heuristics (without aggregation) [11]. Herrmann [16] describes the exchange heuristic in detail.

When considering the waiting time measure, the special greedy (SG) algorithm was used to improve an existing policy. Given a feasible solution, we apply the SG algorithm repeatedly until it converges. Applying the SG algorithm to the heuristic solution generated the HG sequence. Applying the SG algorithm to the AHD sequence generated the AHDG sequence. Under some conditions, the SG algorithm will generate an optimal solution [22]. Herrmann [19] describes the SG algorithm in detail.

Here we report on stride scheduling with $\delta = 0.5$ (this is also known as the Webster heuristic). The results with $\delta = 1$ were very similar. (For details, see [16, 19, 20].)

Before discussing the results of the heuristics, we consider first how many times that an instance could be aggregated. Table 5 shows that the average number of aggregations almost always decreases as $n$ increases. When $T = 100$, the average number of aggregations increases as $n$ increases from 10 to 20. The average number of aggregations per instance is near six for $T = 100$ and $n = 20$, but, as $n$ increases, this decreases to just over two.

As $n$ approaches $T$, the average number of objects in the aggregated instances also decreases because the aggregation depends upon the number of distinct values of $x_i$. Each distinct value leads to an aggregation of multiple objects and generates a new object in the aggregated instance. Thus, the number of objects in the aggregated instance generally equals the number of aggregations needed to create it. Of course, there are some cases in which two new objects can be combined, which increases the number of aggregations and reduces the number of objects, and some objects may have unique values, but this occurred less often as *n* increased. As *n* approaches *T*, the number of distinct values decreases, so there are fewer aggregations and fewer objects in the aggregated instances.

Tables 6, 7, 8, and 9 list the results of the experiments. The measure (count balance, gap balance, RTV, or waiting time) was averaged over the 100 instances in each problem set.

We will first consider the results for minimizing the count balance. As shown in Table 6, using aggregation with stride scheduling reduced the average count balance. As *n* increased, there is a trend towards smaller count balances (because there are more objects that have counts near 1).

As shown in Table 7, the gap balance can be quite large for solutions generated using stride scheduling as *n* increased. In these cases, the large gap balances occur because the small-count objects are all placed together, and multiple copies of the larger-count objects are bunched together, which increases the gap balance of the larger-count objects. Using aggregation with stride scheduling significantly reduced the average gap balance, and the solutions were more balanced as *n* increased because aggregation combined the objects into a small number of high-count objects that were distributed more evenly.

On the RTV performance measure, the results in Table 8 show that the H sequences may perform extremely poorly for large values of *n*. The AHD sequences are much better, especially as *n* increased. As expected, using the exchange heuristic generated better sequences, but aggregation is still useful. Compared with the HE sequences, the AHDE sequences reduce RTV dramatically when *n* is moderate (but not at the extremes). When *n* is near *T*, the AHDE sequences are about the same as the HE sequences because the exchange heuristic is very good at constructing low-RTV sequences in those cases.

As shown in Table 9, the average waiting time in the H sequences generated by stride scheduling are quite large for all values of *n*. The SG algorithm, which constructs the HG policies, reduces average waiting time dramatically when improving these sequences, especially as *n* increased. The quality of the AHD and AHDG sequences is very good compared to the H and HG sequences. The stride scheduling algorithm, which generates poor-quality sequences by itself, performs well when used with aggregation. When *n* is near *T*, all of the AHD sequences have practically no waiting. Compared to the AHD sequences, the AHDG sequences are slightly better. In general, there is little room for improvement because the AHD sequences are quite good.

Table 6. Average values of the count balance for the H and AHD sequences generated using stride scheduling (with $\delta = 0.5$) and aggregation.

| *T* | *n* | H | AHD |
|---|---|---|---|
| 100 | 10 | 2.70 | 2.01 |
| | 20 | 2.85 | 2 |
| | 30 | 3 | 2 |
| | 40 | 3.04 | 1.99 |
| | 50 | 2.80 | 1.95 |
| | 60 | 3.24 | 1.82 |
| | 70 | 3.37 | 1.63 |
| | 80 | 3.08 | 1.58 |
| | 90 | 2.43 | 1.36 |
| 500 | 50 | 3 | 2 |
| | 100 | 3 | 2 |
| | 150 | 3.02 | 2 |
| | 200 | 3.54 | 2 |
| | 250 | 3.71 | 2 |
| | 300 | 3.68 | 1.95 |
| | 350 | 3.79 | 1.92 |
| | 400 | 3.65 | 1.71 |
| | 450 | 2.96 | 1.42 |

Table 7. Average values of the gap balance for the H and AHD sequences generated using stride scheduling (with δ = 0.5) and aggregation.

| *T* | *n* | H | AHD |
|---|---|---|---|
| 100 | 10 | 8.55 | 4.57 |
| | 20 | 16.04 | 3.97 |
| | 30 | 23.70 | 3.32 |
| | 40 | 29.29 | 2.89 |
| | 50 | 34.20 | 2.65 |
| | 60 | 49.46 | 2.16 |
| | 70 | 64.16 | 1.80 |
| | 80 | 75.74 | 1.58 |
| | 90 | 83.80 | 0.79 |
| 500 | 50 | 45.31 | 7.28 |
| | 100 | 84.33 | 5.89 |
| | 150 | 117.60 | 5.09 |
| | 200 | 150.47 | 4.16 |
| | 250 | 179.02 | 3.49 |
| | 300 | 249.33 | 2.91 |
| | 350 | 322.94 | 2.52 |
| | 400 | 388.71 | 1.92 |
| | 450 | 443.66 | 1.34 |

Table 8. Average values of RTV for the H, HE, AHD, and AHDE sequences generated using stride scheduling (with δ = 0.5), aggregation, and the exchange heuristic.

| *T* | *n* | H | HE | AHD | AHDE |
|---|---|---|---|---|---|
| 100 | 10 | 262.5 | 98.3 | 95.9 | 73.0 |
| | 20 | 1,016.6 | 165.7 | 82.4 | 59.1 |
| | 30 | 2,548.8 | 180.2 | 60.7 | 39.0 |
| | 40 | 4,662.1 | 217.6 | 47.8 | 26.1 |
| | 50 | 8,540.9 | 85.6 | 39.5 | 18.3 |
| | 60 | 16,072.1 | 36.3 | 25.9 | 9.0 |
| | 70 | 26,778.2 | 5.3 | 14.0 | 3.7 |
| | 80 | 35,352.1 | 1.4 | 10.1 | 1.3 |
| | 90 | 31,222.4 | 0.3 | 1.8 | 0.3 |
| 500 | 50 | 22,159.6 | 2,006.0 | 862.8 | 513.6 |
| | 100 | 111,484.2 | 2,099.7 | 590.2 | 306.3 |
| | 150 | 285,794.8 | 6,778.6 | 434.8 | 211.6 |
| | 200 | 561,918.5 | 1,850.0 | 315.3 | 153.0 |
| | 250 | 1,056,653.4 | 650.9 | 212.8 | 83.0 |
| | 300 | 2,012,186.5 | 415.9 | 152.9 | 42.1 |
| | 350 | 3,358,079.2 | 25.1 | 102.3 | 17.7 |
| | 400 | 4,421,555.9 | 7.7 | 51.0 | 6.5 |
| | 450 | 3,908,763.9 | 1.5 | 20.6 | 1.5 |

Table 9. Average values of waiting time for the H, HG, AHD, and AHDG sequences generated using stride scheduling (with δ = 0.5), aggregation, and the SG algorithm.

| *T* | *n* | H | HG | AHD | AHDG |
|---|---|---|---|---|---|
| 100 | 10 | 1.70 | 1.25 | 1.00 | 0.94 |
| | 20 | 2.65 | 1.59 | 0.70 | 0.65 |
| | 30 | 3.56 | 1.88 | 0.48 | 0.44 |
| | 40 | 4.58 | 1.91 | 0.36 | 0.32 |
| | 50 | 5.65 | 1.99 | 0.27 | 0.25 |
| | 60 | 6.72 | 1.28 | 0.18 | 0.16 |
| | 70 | 7.10 | 0.60 | 0.10 | 0.09 |
| | 80 | 6.34 | 0.22 | 0.06 | 0.06 |
| | 90 | 4.04 | 0.10 | 0.01 | 0.01 |
| 500 | 50 | 7.49 | 4.91 | 1.35 | 1.20 |
| | 100 | 12.88 | 6.23 | 0.85 | 0.71 |
| | 150 | 17.16 | 8.60 | 0.59 | 0.50 |
| | 200 | 22.64 | 8.90 | 0.42 | 0.37 |
| | 250 | 28.64 | 7.59 | 0.29 | 0.25 |
| | 300 | 33.85 | 4.95 | 0.20 | 0.18 |
| | 350 | 35.57 | 1.96 | 0.13 | 0.12 |
| | 400 | 31.73 | 0.34 | 0.06 | 0.06 |
| | 450 | 20.21 | 0.10 | 0.02 | 0.02 |

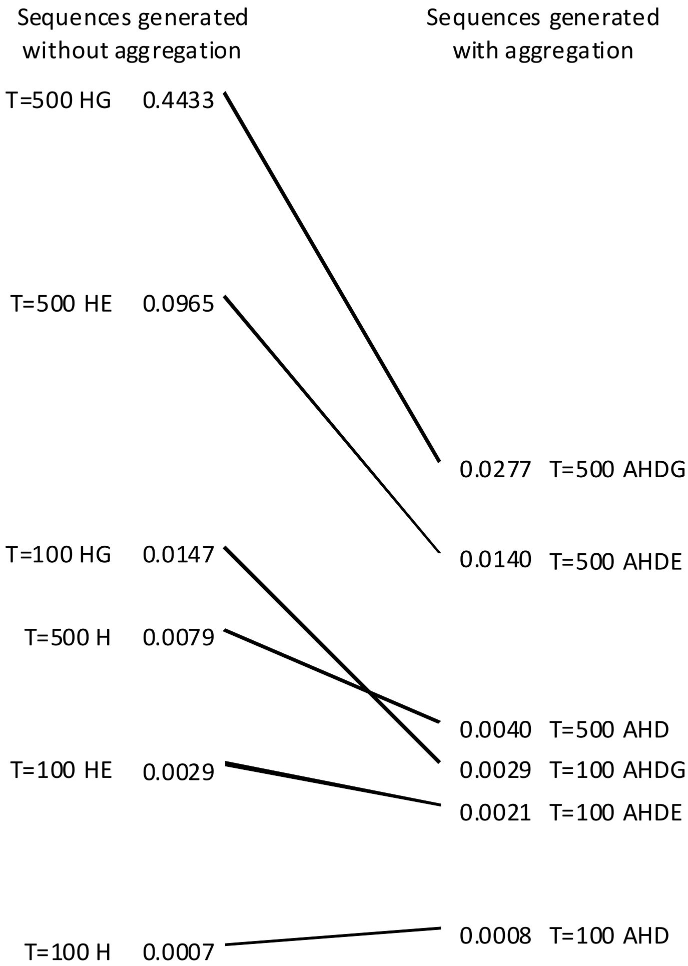


Figure 2. Average time (seconds) required to generate sequences.
Results are averaged over all instances with the given value of *T*.
Note that the vertical axis is logarithmic to improve readability.

To understand these results, consider a set of objects with the same count. The stride scheduling algorithm will put copies of these objects next to each other, which leads to poor-quality sequences, especially as the number of objects *n* approaches the number of positions *T*. Aggregation groups these objects, and the positions assigned to the new objects by stride scheduling are distributed across the aggregated sequence. The positions assigned to this set of objects in the disaggregated sequence are also distributed, leaving space for the objects with higher counts. As *n* approaches *T*, the sequences formed using aggregation are nearly ideal.

This explanation hints that aggregation would not perform as well if other heuristics were used. The results in [16, 18, 19, 20] show, however, that, when used with other heuristics, including those that perform better than stride scheduling, aggregation finds sequences that are more fair than those generated without using aggregation. In addition, aggregation is robust in the sense that changing the sequencing algorithm may make little difference in the quality of the sequences generated. In this paper, we emphasize the results with stride scheduling because it has been used on all of these problems and in other work [7, 11, 13].

We also measured the clock time needed to generate these policies. Figure 2 shows the average time needed to generate the different solutions for different heuristics and different values of *T*. These are averages over all of the corresponding problem sets. As *T* increased, the time required increased for all heuristics.

Compared with the time required to generate the H sequences, using aggregation with stride scheduling to generate the AHD sequences required more time when $T$ = 100 but required less time when $T$ = 500. (Using aggregation also required less time when *T* was larger [19].) Reducing the number of objects by aggregation (from hundreds to around ten) reduces the computational effort of the stride scheduling algorithm. The extra effort of aggregating and disaggregating did not add much time for the larger instances.

Because the H sequences had very large values of RTV, generating the HE sequences (to reduce RTV) required a great deal of time, but improving the AHD sequences to generate the AHDE sequences required less additional time. Overall, generating the AHDE sequences required less time than generating the HE sequences.

Generating the HG solutions (to improve average waiting time) requires more time than generating the H solutions. The quality of the HG solutions is significantly better than the quality of the H solutions for some values of *n*, which shows that the extra effort generated some benefit. Although improving the AHD sequences to generate the AHDG sequences required additional time, generating the AHDG sequences required less time than generating the HG sequences.

Overall, these results show that using aggregation with the stride scheduling algorithm generates sequences that are more fair (measured with different metrics) with less computational effort. In particular, the sequences are more balanced (they have lower count-balances and gap-balances), they have smaller values for RTV, and they have less waiting time.

## 9 Summary and Conclusions

This paper presents an aggregation approach for generating fair sequences. We showed that using this approach with the well-known stride scheduling algorithm generates better solutions more quickly. This results holds for different measures of fairness, including count balance, gap balance, RTV, and average waiting time and for a range of problem sizes.

The aggregation approach presented here should be effective at generating good sequences for other fair sequencing problems in which the objective is to minimize the deviation from a perfectly fair sequence, including scheduling just-in-time manufacturing [9] and the bottleneck problem [13]. Problems in which one must satisfy hard constraints on the deviations (like the periodic maintenance scheduling problem and the Liu-Layland problem) will require a different approach, though aggregation could be used to accelerate the search for feasible solutions.

**Acknowledgements** This work was originally motivated by a collaboration with Leonard Taylor at the University of Maryland Medical Center.

## References

1. Kubiak, W., Fair sequences, in Handbook of Scheduling: Algorithms, Models and Performance Analysis, Leung, J.Y-T., editor, Chapman & Hall/CRC, Boca Raton, Florida, 2004.

2. Bar-Noy, Amotz, Randeep Bhatia, Joseph Naor, and Baruch Schieber, Minimizing service an operation costs of periodic scheduling, Mathematics of Operations Research, 27, 518-544, 2002.
3. Bar-Noy, Amotz, Aviv Nisgah, and Boaz Patt-Shamir, Nearly Optimal Perfectly Periodic Schedules, Distributed Computing, 15, 207-220, 2002.
4. Wei, W.D., and Liu, C.L., On a Periodic Maintenance Problem, Operations Research Letters, 2, 90-93, 1983.
5. Park, Kyung S., and Doek K. Yun, Optimal Scheduling of Periodic Activities, Operations Research, 33, 690-695, 1985.
6. Campbell, Ann Melissa, and Jill R. Hardin, Vehicle Minimization for Periodic Deliveries, European Journal of Operational Research, 165, 668–684, 2005.
7. Waldspurger, C.A., and Weihl, W.E., Stride Scheduling: Deterministic Proportional-Share Resource Management, Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1995.
8. Balinski, M.L, and H.P. Young, Fair Representation, Yale University Press, New Haven, Connecticut, 1982.
9. Miltenburg, J., Level Schedules for Mixed-Model Assembly Lines in Just-in-Time Production Systems, Management Science, 35, 192-207, 1989.
10. Inman, R.R., and Bulfin, R.L., Sequencing JIT Mixed-Model Assembly Lines, Management Science, 37, 901-904, 1991.
11. Corominas, Albert, Wieslaw Kubiak, and Natalia Moreno Palli, Response Time Variability, Journal of Scheduling, 10, 97-110, 2007.
12. Garcia, A., R. Pastor, and A. Corominas, Solving the Response Time Variability Problem by Means of Metaheuristics, in Artificial Intelligence Research and Development, edited by Monique Polit, T. Talbert, and B. Lopez, pages 187-194, IOS Press, 2006.
13. Kubiak, Wieslaw, Proportional Optimization and Fairness, Springer, New York, 2009.
14. Rogers, David F., Robert D. Plante, Richard T. Wong, and James R. Evans, Aggregation and Disaggregation Techniques and Methodology in Optimization, Operations Research, 39, 553-582, 1991.
15. Herrmann, Jeffrey W., Generating Cyclic Fair Sequences using Aggregation and Stride Scheduling, Technical Report 2007-12, Institute for Systems Research, University of Maryland, College Park, 2007.
16. Herrmann, Jeffrey W., Using Aggregation to Reduce Response Time Variability in Cyclic Fair Sequences, to appear in Journal of Scheduling. DOI: 10.1007/s10951-009-0127-7. Published online on August 15, 2010.
17. Herrmann, Jeffrey W., Constructing Perfect Aggregations to Eliminate Response Time Variability in Cyclic Fair Sequences, Technical Report 2008-29, Institute for Systems Research, University of Maryland, October, 2008.
18. Herrmann, Jeffrey W., Generating Cyclic Fair Sequences for Multiple Servers, Proceedings of the 4th Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA 2009), Dublin, Ireland, August 10-12, 2009.
19. Herrmann, Jeffrey W., Using Aggregation to Construct Periodic Policies for Routing Jobs to Parallel Servers with Deterministic Service Times, to appear in Journal of Scheduling. DOI: 10.1007/s10951-010-0209-6. Published online on November 24, 2010.
20. Herrmann, Jeffrey W., Aggregating Alphabets to Construct Balanced Words, Technical Report 2009-12, Institute for Systems Research, University of Maryland, September, 2009.
21. Sano, Shinya, Naoto Miyoshi, and Ryohei Kataoka, m balanced words: A generalization of balanced words, Theoretical Computer Science, Volume 314, Issues 1-2, 25 February 2004, pages 97-120, 2004.
22. van der Laan, D.A., Routing jobs to servers with deterministic service times, Mathematics of Operations Research, 30(1), 195-224, 2005.