MISTA 2009

Generating Cyclic Fair Sequences for Multiple Servers

Jeffrey W. Herrmann

Abstract Fair sequences allocate capacity to competing demands in a variety of manufacturing and computer systems. This paper considers the generation of cyclic fair sequences for a set of servers that must process a given set of products, each of which must be produced multiple times in each cycle. The objective is to create a sequence that minimizes, for each product, the variability of the time between consecutive completions (the response time). Because minimizing response time variability is known to be NP-hard and the performance of existing heuristics is poor for certain classes of problems, we present an aggregation approach that 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. Computational experiments show that using aggregation can reduce computational effort and response time variability dramatically.

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 is no single definition 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

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

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.

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, using 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 problem and 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.

Independently, Herrmann [14] 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 [15] 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 [16] considered a more sophisticated "perfect aggregation" approach that can generate zero RTV sequences for some instances.

In this work we consider the problem when multiple servers, working in parallel, are available. Although the problem has some similarities to the problem of scheduling real-time tasks on multiprocessors, in the problem studied here, we will generate a complete schedule for the entire cycle (instead of relying on priority rules to determine which task to perform at each point in time), and our objective is to reduce response time variability (not to meet time windows on each subtask). The purpose of this paper is to determine how the aggregation technique performs in combination with heuristics for the multiple-server RTV problem. The current paper thus extends previous work on the RTV problem (which considered only a single server).

The remainder of this paper describes the problem formulation, presents the aggregation approach, discusses the heuristics and a lower bound, and presents the results of extensive computational experiments using the aggregation approach in combination with heuristics developed for this problem.

2 Problem Formulation

Given M servers that must produce n products, each with a demand d_i that is a positive integer, let $D = d_1 + \cdots + d_n$. (We assume that M divides D, as we can add sufficient dummy products with $d_i = 1$ to pad the instance.) Each product requires the same amount of time, so we can ignore time and consider only the positions in the sequence. A feasible sequence $s = s_1 s_2 \ldots s_L$ has length $L = D/M$ and specifies the M products that are served in each of the L positions. That is, for $j = 1, \ldots, L$, each s_j is a subset of $\{1, \ldots, n\}$ with exactly M elements. Each product i occurs exactly d_i times in the sequence. No product can be processed by two servers at the same time. (Thus, all $d_i \le L$.) Moreover, this sequence will be repeated, and we will call each occurrence a cycle. The RTV of a feasible sequence equals the sum of the RTV for each product. If product 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 $p_{i0} = p_{id_i} - L$). The average interval for product i is L/d_i, so we calculate RTV as follows:

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

Thus, we can describe the problem as follows: Given an instance $\{d_1, \ldots, d_n\}$, find a sequence s of length L that minimizes RTV subject to the constraints that exactly M different products are served in each and every one of the L positions and each and every product i occurs exactly d_i times in s.

The single-server version of this problem is NP-hard [11], and it is easy to see that the multiple-server version must be NP-hard as well.

Consider, as an example, the following nine-product instance: {1, 1, 2, 2, 2, 2, 2, 3, 5}. Table 1 lists a feasible sequence for this instance; its RTV equals 16.67.

Table 1. A feasible sequence for a two-server, nine-product RTV problem.
(The product demands are {1, 1, 2, 2, 2, 2, 2, 3, 5}.)

Server										
1	9	3	5	7	1	8	9	4	6	8
2	8	4	6	9	2	9	3	5	7	9

3 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 [17]. 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 single-server RTV problem, iteratively combining products with the same demand transforms the original instance into one with fewer products. This aggregation can lead to dramatically better solutions [15]. This was first introduced in [14] and is similar to the substitution concept discussed by [4]. This paper presents a version of this aggregation that is appropriate for multiple servers.

For a set of products that all have the same demand, the positions in a sequence where those products occur should be assigned to those products in a round-robin fashion. Thus, for that set of products, the challenge of assigning positions to those products is equivalent to assigning positions to the set as a whole. Once an appropriate number of positions has been assigned to the entire set, it is easy to determine which products should go in which positions.

The aggregation combines products that have the same demand into groups. This has two benefits. First, it reduces the number of products that need to be sequenced. Secondly, because a group will have higher demand, it will be blended better than the individual products. Nevertheless, the demand of a group must be no greater than the cycle length L.

For instance, consider the nine-product example presented above. Combining the products 3 to 7, which all have a demand of 2, into one group with a demand of 10, simplifies the problem significantly. Because the sequence length equals 10, it is clear that this group should be served in every position. From there, it is a straightforward task to assign positions 1 and 6 to one of the five products, positions 2 and 7 to a second product, and so forth.

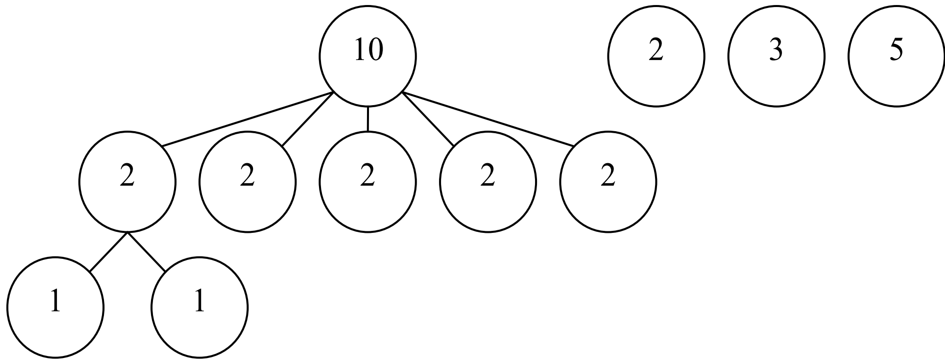

Figure 1. The forest of weighted trees corresponding to the aggregation of the two-server, nine-product example. The number in each node is the demand of the corresponding product.

The aggregation approach presented here repeatedly aggregates an instance until it cannot be aggregated any more. In our example, the first aggregation combines the first two products into a group with a demand of 2. This generates a new instance that has six products with a demand of 2. We cannot combine all six products because the sequence length equals only 10.

Thus, the second aggregation combines five to these products to form a group with a demand of 10. This generates a second new instance that has only four products, and their demands are {2, 3, 5, 10}. Because they all have different demands, no more aggregation is possible. As discussed in more detail below, an aggregation can be visualized as a forest, as shown in Figure 1.

The aggregation procedure generates a sequence of instances I_0, …, I_H. (H is the index of the last aggregation created.) The aggregation can be done at most $n_0 - 1$ times because the number of products decreases by at least one each time the aggregation procedure is called (unless no aggregation occurs). Thus $H \le n_0 - 1$.

The more complex notation used in the following steps 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 products in instance I_k. Let P_j be the set of products that form the new product j, and let $P_j(i)$ be the i-th product in that set.

Aggregation. Given: an instance I_0 with demands $\{d_1, \ldots, d_n\}$ and M servers.

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

3a. Let i be one of the products in G. If $|G| > L/d_i$, then G is too large. Keep $\lfloor L/d_i \rfloor$ products and remove any others.

3b. Let $m = |G|$. Create a new product $n + k + 1$ with demand $d_{n+k+1} = md_i$. Create the new instance I_{k+1} by removing from I_k all m products in G and adding product $n + k + 1$. Set $P_{n+k+1} = G$ and $n_{k+1} = n_k - m + 1$. Increase k by 1 and go to Step 2.

Consider the nine-product example presented above. The first aggregation creates instance I_1 by combining two products of I_0 (G = {1, 2}) and creating product 10 with $P_{10} = \{1, 2\}$ and $d_{10} = 2$. The new instance has $n_1 = 8$ products.

The second aggregation creates instance I_2 by combining the five products of I_1 (G = {3, 4, 5, 6, 10}) and creating product 11 with P_{11} = {3, 4, 5, 6, 10} and $d_{11} = 10$. (Note that product 7 cannot be included due to the length constraint.) The new instance has $n_2 = 4$ products. No more aggregation is possible, so $H = 2$. See also Table 2 and Figure 1.

Table 2. The product sets for the original and new instances.

	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}
I_0	1	1	2	2	2	2	2	3	5		
I_1			2	2	2	2	2	3	5	2	
I_2							2	3	5		10

The total demand in each new instance will equal the total demand of the original instance because the demand of the new product (which we call a "group") equals the total demand of the products that were removed.

We can represent the aggregation as a forest of weighted trees (as shown in Figure 1). There is one tree for each product in I_H. The weight of the root of each tree is the total demand of the products that were aggregated to form the corresponding product 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 products in I_0 that were aggregated to form the corresponding product in I_0, and each one's weight equals the demand of that product. The forest has one parent node for each group formed during the aggregation, and the total number of nodes in the forest equals $n_0 + H < 2n_0$.

Let S_H be a feasible sequence for instance I_H. Disaggregating S_H requires H steps that correspond to the aggregations that created the new instances. 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 is to replace each product from that instance with the corresponding product (or products) from the less-aggregate instance. Consider, for instance, the example presented above and S_2, a feasible sequence for I_2, shown as the first sequence in Table 3. Because product 11 in I_2 was formed by combining five products in I_1 (the products in P_{11}), the 10 positions in S_2 that are assigned to that product will be given to those five products. This disaggregation yields the sequence S_1, a feasible sequence for I_1, shown as the second sequence in Table 3.

This sequence, in turn, can be disaggregated to form S_0, a feasible sequence for I_0, shown as the third sequence in Table 3. Because product 10 in I_1 was formed by combining products 1 and 2 in I_0, the positions in S_1 that are assigned to that product will be given to those two products.

Table 3. Initial and disaggregated solutions.

Server										
1	11	11	11	11	11	11	11	11	11	11
2	9	7	9	8	9	7	9	8	9	8
1	3	4	5	6	10	3	4	5	6	10
2	9	7	9	8	9	7	9	8	9	8
1	3	4	5	6	1	3	4	5	6	2
2	9	7	9	8	9	7	9	8	9	8

In the following $j = S_k(s,a)$ means that server s processes product j in position a of sequence S_k. $P_{n+k}(i)$ is the i-th product in P_{n+k}.

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

1. Initialization. Let $k = H$.
2. Set $m = |P_{n+k}|$ and $i = 1$.
3. For $a = 1,\ldots,L$, perform the following step:
   a. For $s = 1,\ldots,M$, perform the following step:
      i. Let $j = S_k(s,a)$. If $j < n+k$, then assign $S_{k-1}(s,a) = j$. Otherwise, assign $S_{k-1}(s,a) = P_{n+k}(i)$, 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.

Aggregation runs in O(n^2) time because each aggregation requires O(n) time and there are at most $n-1$ aggregations. Likewise, because each sequence disaggregation requires O(D) effort, disaggregation runs in O(nD) time in total.

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 products in a group may have unequal demands and (2) the placement of products in the tree is not specified. Because our aggregation scheme groups products with equal demand, the disaggregation is much simpler. The limitation, however, is that the instance must have some equal demand products.

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.

4 Heuristics

The purpose of this paper is to investigate how much the aggregation approach reduces RTV when used with various heuristics for the multi-server problem. This section presents the heuristics that will be used. There are four basic heuristics (which construct sequences) and an improvement heuristic that manipulates sequences.

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. Here we describe a multiple-server version of the algorithm.

Parameterized stride scheduling. Given: an instance $\{d_1, \ldots, d_n\}$, M servers, and the parameter δ.

1. Initialization. $x_{i0} = 0$ for i = 1, …, n. (x_{ik} is the number of times that product i occurs in the first k positions of the sequence.)
2. For k = 0, …, D-1, perform the following steps:
   a. For all products i that satisfy $d_i - x_{ik} = L - k$, assign those products to servers in position $k+1$ (because the remaining demand for the product equals the number of remaining positions in the sequence).
   b. Any unassigned servers are assigned the products with the greatest values of the ratio $d_i / (x_{ik} + \delta)$. In case of a tie, select the lowest-numbered product to break a tie.
   c. For all products i that were assigned to a server, set $x_{i,k+1} = x_{i,k} + 1$. For all other products, set $x_{i,k+1} = x_{i,k}$.

The computational effort of the algorithm is O(nD). Table 1 shows the solution generated by the parameterized stride scheduling algorithm (with δ = 0.5) for the nine-product instance introduced earlier. The RTV equals 16.67. (Other values of δ change the relative positions of some products but do not affect the RTV.) Note that products 3 to 7 (which all have the same demand) are grouped together first in positions 2 to 4 and again in positions 7 to 9, which leads to high RTV for the higher-demand products.

Two versions of the parameterized stride scheduling algorithm are well-known apportionment methods. Following [11], we will use the names “Webster” and “Jefferson” to denote these heuristics. 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. Clearly, both the Webster and Jefferson heuristics run in O(nD) time.

The bottleneck heuristic is the due date algorithm that Steiner and Yeomans [18] developed for solving the mixed-model production problem, modified to schedule multiple servers. This heuristic runs in O(D log D) time. Here we will describe the algorithm in detail.

Bottleneck. Given: an instance $\{d_1,\ldots,d_n\}$ and M servers.

1. Set $w = 0$ and $x_i = 1$ for all $i = 1,\ldots,n$.
2. For $i = 1,\ldots,n$ and $j = 1,\ldots,d_i$, calculate the following quantities:

$$EST_{ij} = \left\lceil \frac{L(j-1)+w}{d_i} - 1 \right\rceil$$

$$LST_{ij} = \left\lfloor \frac{Lj - w}{d_i} \right\rfloor$$

3. For $k = 0,\ldots,L-1$, perform the following steps:
   a. Let $R = \left\{ i : x_i \le d_i, EST_{ix_i} \le k, LST_{ix_i} \ge k \right\}$.
   b. If $|R| \ge M$, then let R' be the M products in R that have the smallest LST_{ix_i}. Otherwise, go to Step 5.
   c. For all i in R', assign one task of product i to position k + 1 on one server, and add one to x_i.
4. Save the current sequence. If $w < \max\{d_i\}$, then set w to the value of the smallest d_i that is greater than w, and go to Step 2. Otherwise, go to Step 5.
5. Return the last saved sequence.

The sequential heuristic is an extremely simple sequence construction approach. It builds the sequence one product at a time, assigning all of the positions for one server and then moving to the next one. (It is similar to scheduling preemptive tasks on parallel machines.)

Sequential. Given: an instance $\{d_1,\ldots,d_n\}$ and M servers.

1. Set s = 1 and a = 1.
2. For $i = 1,\ldots,n$, perform the following step:
   a. For $j = 1,\ldots,d_i$ perform the following steps:
      i. Assign product i to server s in position a.
      ii. Increase a by 1. If $a > L$, set $s = s + 1$ and a = 1.

This heuristic runs in O(D) time. Table 5 shows the solution generated by the sequential for the nine-product instance introduced earlier. The RTV equals 212.67.

Table 5. Solution generated by the sequential heuristic.

Server										
1	1	2	3	3	4	4	5	5	6	6
2	7	7	8	8	8	9	9	9	9	9

The exchange heuristic is a technique for improving any given sequence [11]. It repeatedly loops through the positions, exchanging a task with the task immediately following it if that exchange reduces the RTV or reduces the maximum distance between two tasks for a product. It runs in O(nD^4) time. We modified the exchange heuristic for the multiple server case by restricting exchanges to those that strictly reduce the RTV and finding, among all possible exchanges between two consecutive positions, the one that most reduces RTV.

Exchange. Given: an instance $\{d_1,\ldots,d_n\}$, M servers, and a sequence S.

1. Set *flag* = 0.
2. For a = 1, …, L, perform the following steps:
   a. If $a < L$, set $b = a$+1. Otherwise, set b = 1.
   b. For s = 1, …, M, perform the following step:

i. Set $i = S(s, a)$. Set L_i equal to the distance between position a and the last position in which product i occurs before position a (going back to the last cycle if needed). Set R_i equal to the distance between position a and the next position in which product i occurs after position a (going on to the next cycle if needed). If $d_i = 1$, set $E_i = 0$; otherwise, set $E_i = 2(R_i - L_i - 1)$, which is the decrease in product i's RTV if product i is moved to position b. If product i already occurs in position b, then this move is not feasible.

c. Find the product i^* among those that occur in position a that gives the largest value of E_i and can be moved feasibly to position b.

d. For $s = 1, \ldots, M$, perform the following step:

i. Set $j = S(s, b)$. Set L_j equal to the distance between position b and the last position in which product j occurs before position b (going back to the last cycle if needed). Set R_j equal to the distance between position b and the next position in which product j occurs after position b (going on to the next cycle if needed). If $d_j = 1$, set $F_j = 0$; otherwise, set $F_j = 2(L_j - R_j - 1)$, which is the decrease in product j's RTV if product j is moved to position a. If product j already occurs in position a, then this move is not feasible.

e. Find the product j^* among those that occur in position b that gives the largest value of F_j and can be moved feasibly to position a.

f. If $E_{i^*} + F_{j^*} > 0$, then exchange i^* and j^* by moving i^* to position b and moving j^* to position a, and set *flag* = 1.

3. If *flag* = 1, then go to Step 1. Otherwise, stop and return the current sequence.

For instance, consider our nine-product example and the sequence shown in Table 5. Suppose that the exchange heuristic is considering position 1. When $a = 1$, product 7 already occurs in position 2. Thus, $i^* = 1$. Likewise, $j^* = 2$. $E_1 + F_2 = 0 + 0 = 0$, so no exchange occurs. So the heuristic moves to position $a = 2$. Here, $E_2 = 0$ and $E_7 = 2(9\text{-}1\text{-}1) = 14$, so $i^* = 7$. With $b = 3$, $F_3 = 2(9\text{-}1\text{-}1) = 14$ and $F_8 = 2(8\text{-}1\text{-}1) = 12$, so $j^* = 3$. Because $E_{i^*} + F_{j^*} > 0$, the heuristic exchanges products 7 and 3 and then goes on to $a = 3$.

5 Lower Bound

To provide some idea of the quality of the solutions generated, we will compare their RTV to a lower bound. The lower bound on the total RTV is the sum of lower bounds for each product's RTV. This lower bound depends on product i's demand and the total sequence length L. Let LB_i be the lower bound on the RTV of product i, which can be calculated as follows by considering intervals that are just shorter and longer than the average:

1. Set $a_i = L \bmod d_i$ (the number of intervals that will be longer than the average) and $A_i = \lfloor L/d_i \rfloor$ (the largest integer less than or equal to L/d_i, which equals the length of the shorter interval) and $B_i = A_i + 1$ (the length of the longer interval).
2. Set $LB_i = (d_i - a_i)(A_i - L/d_i)^2 + a_i(B_i - L/d_i)^2$.

We note that, if d_i divides L, $LB_i = 0$. The lower bound LB on the total RTV is the sum of the product lower bounds: $LB = LB_1 + \cdots + LB_n$.

6 Computational Experiments

The purpose of the computational experiments was to determine how the aggregation technique performs in combination with a variety of heuristics on the metric of RTV. All of the algorithms were implemented in Matlab and executed using Matlab R2006b on a Dell Optiplex GX745 with Intel Core2Duo CPU 6600 @ 2.40 GHz and 2.00 GB RAM running Microsoft Windows XP Professional Version 2002 Service Pack 3.

We considered two values for the number of servers: M = 5 and M = 10. We used 3900 of the 4700 instances described in [15]. These were generated as follows: First, we set the total number of tasks D and the number of products n. (We followed [11] in choosing values of D and n, but these are not the same instances.) To generate an instance, we generated $D – n$ random numbers from a discrete uniform distribution over {1, …, n}. We then let d_i equal 1 plus the number of copies of i in the set of $D – n$ random numbers (this ensured that each d_i is at least 1). We generated 100 instances for each problem set (a combination of D and n). In this study, we used the following problem sets: with D = 100, n = 40, 50, 60, 70, 80, and 90; with D = 500, n = 50, 100, 150, 200, 250, 300, 350, 400, and 450; with D = 1000, n = 50, 100, 200, 300, 400, 500, 600, 700, 800, and 900; and with D = 1500, n = 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, and 1400.

For each instance and value of M, we constructed 16 sequences as follows. First, we applied one of the basic heuristics to the instance (we call this the H sequence). Then, we applied the exchange heuristic to the H sequence to construct the HE sequence.

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 sequence. Then, we applied the exchange heuristic to the AHD sequence to construct the AHDE sequence. This makes four sequences using one basic heuristic and combinations of aggregation-disaggregation and the exchange heuristic. We repeated this for the remaining basic heuristics for a total of 16 solutions.

Before discussing the results of the heuristics, we note that all of the instances considered could be aggregated at least once. For a given value of D, the average number of times that an instance was aggregated increased as D increased but decreased as the number of products n increased. For M = 5, D = 100, and n = 90, the average number of aggregations was 5.1. For M = 5, D = 1500, and n = 100, the average number of aggregations was 14.8.

For M = 10, the average number of aggregations exhibited the same trends, and ranged from 10.1 (with D = 100 and n = 90) to 17.3 (with D = 1500 and n = 100). More aggregations were possible due to the shorter sequence length.

Compared to the single-server problem [15], the number of aggregations is generally larger because the cycle length restriction limits the number of products that can be combined into a group. That is, instead of combining many products with the same demand into one large group, the aggregation must form multiple groups that are not too long, and this increases the number of aggregations.

Because the exchange heuristic is known to reduce RTV, our discussion will focus on the HE and AHDE sequences. Tables 6 to 9 present the results for the HE and AHDE sequences for each of the basic heuristics when M = 5. Tables 10 to 13 present the results for the HE and AHDE sequences for each of the basic heuristics when M = 10. The results are averaged over all 100 instances in each set.

Using aggregation with the Webster, Jefferson, and bottleneck heuristics generates the best sequences. Compared to the HE sequences, the AHDE sequences reduce RTV dramatically when n is small or moderate. For instance, when n = 300 and D = 1000, the average RTV of the AHDE sequences is less than 10% of the average RTV of the HE sequences. When n is near D, the quality of the AHDE sequences is about the same as the quality of the HE sequences because the exchange heuristic is very good at constructing low-RTV sequences in those cases.

The sequential heuristic performs poorly overall. When n is small, the RTV of the HE and AHDE sequences exceeds the RTV of the HE and AHDE sequences generated using the other heuristics. When n is moderate, the RTV of the HE sequences is better (compared to the other HE sequences), but the RTV of the AHDE sequences is larger (compared to the other AHDE sequences). When n is large, the RTV of the HE and AHDE sequences is very low and close to the RTV of the HE and AHDE sequences generated using the other heuristics.

Table 14 lists the average lower bound for the instances in each set for both $M = 5$ and $M = 10$. When the number of machines is larger, the sequence length is shorter, which increases the RTV of a product because it is harder to spread out the production evenly. As n increases, the lower bounds decrease because the products have lower demand and are easier to distribute evenly. The RTV of the AHDE sequences generated using the Webster, Jefferson, and bottleneck heuristics is much closer to the lower bound than that of the HE sequences. For instance, when $M = 5$, $D = 1000$, and $n = 600$, the average RTV of the HE sequences generated using the Webster heuristic equals 100.4, the average RTV of the AHDE sequences equals 50.6, and the average lower bound equals 46.1, which shows that the AHDE sequences are near-optimal for these instances. For $M = 10$, the largest average gap between the lower bound and the RTV of the AHDE sequences generated using the Webster heuristic equals 9.7 when $D = 1500$ and $n = 200$. For this same set, the average gap between the lower bound and the RTV of the HE sequences generated using the Webster heuristic equals 1470.8.

The computational effort required to generate these sequences depended greatly upon D, the heuristic, and the type of sequence (see Figure 2). The computational effort increased significantly as D increased. For a given value of D, the following sequences took the most time: generating the H and HE sequences with the bottleneck heuristic (around 16 seconds per instance when $D = 1500$ and $M = 5$), generating the HE with the sequential heuristic (around 3 seconds when $D = 1500$ and $M = 5$), generating the HE sequences with the Webster and Jefferson heuristics (around 2 seconds when $D = 1500$ and $M = 5$), and generating the AHDE sequences with the sequential heuristic (just over 1 second when $D = 1500$ and $M = 5$). Generating the H sequences with the sequential heuristic took the least time by far, with average times around 0.0003 seconds when $D = 1500$ and $M = 5$.

For the Webster, Jefferson, and bottleneck heuristics, generating the AHDE sequences took dramatically less time than generating the HE sequences. For the Webster and Jefferson heuristics, with $M = 5$ and $D = 1500$, the average time needed to generate an AHDE sequence was less than 4% of the time needed to generate an HE sequence (0.0632 seconds to 1.6835 seconds). For the bottleneck heuristic, with $M = 5$ and $D = 1500$, the average time needed to generate an AHDE sequence was less than 1% of the time needed to generate an HE sequence (0.1524 seconds to 16.1263 seconds). Across all values of M, D, and n, generating AHDE sequences required less time with the Webster and Jefferson heuristics than with the bottleneck heuristic (using the Webster heuristic or Jefferson heuristic reduced the time by 15 to 80%).

In some cases, the computational effort required changed as n increased, but the magnitude of these changes was less significant than the changes due to changing other parameters. More time was required to construct the sequences with $M = 5$ than with $M = 10$ due to the longer sequence length. Generally, it took about twice as long to construct a sequence. None of the trends mentioned above changed with the increase in M.

Table 6. Comparison of the HE and AHDE sequences across four basic heuristics for $M = 5$ and $D = 100$.

	Webster		Jefferson		Bottleneck		Sequential	
n	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
40	47.7	8.5	46.1	8.4	53.4	8.5	20.8	16.2
50	15.6	7.2	17.7	7.0	19.6	7.2	9.3	9.4
60	5.5	4.8	8.0	4.8	7.8	4.9	6.6	6.0
70	3.4	2.9	3.3	3.1	3.6	3.0	3.5	3.4
80	1.5	1.3	1.3	1.3	1.3	1.4	1.3	1.6
90	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3

Table 7. Comparison of the HE and AHDE sequences across four basic heuristics for $M = 5$ and $D = 500$.

	Webster		Jefferson		Bottleneck		Sequential	
n	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
50	364.7	88.6	357.6	90.4	360.1	90.9	540.9	437.8
100	472.3	73.5	470.0	72.8	460.9	75.0	209.6	246.7
150	603.8	52.2	564.9	52.5	552.7	55.4	349.5	308.9
200	281.2	46.1	264.6	45.7	336.6	46.0	140.2	138.6
250	89.7	36.0	99.1	36.1	160.1	37.7	57.5	55.7
300	40.4	25.7	47.3	25.5	128.8	26.9	34.7	32.1
350	17.4	14.7	15.9	14.8	16.9	15.6	16.9	15.7
400	8.0	6.5	8.0	6.4	7.0	6.5	6.7	7.5
450	1.6	1.5	1.6	1.5	1.5	1.5	1.5	1.5

Table 8. Comparison of the HE and AHDE sequences across four basic heuristics for $M = 5$ and $D = 1000$.

	Webster		Jefferson		Bottleneck		Sequential	
n	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
50	577.1	206.5	574.2	211.8	579.6	206.6	3395.5	2885.6
100	1292.5	182.9	1291.2	185.8	1231.2	183.8	1150.8	993.0
200	1373.5	141.9	1176.1	141.7	1096.5	145.1	437.1	680.9
300	2961.6	106.0	1751.9	106.9	3077.1	110.6	994.1	1021.1
400	666.3	91.8	516.7	91.5	717.3	90.3	319.1	323.0
500	211.1	72.4	215.1	73.9	237.9	75.0	127.2	127.7
600	100.4	50.6	116.4	50.0	253.7	54.8	71.0	61.4
700	39.6	29.9	34.0	29.5	38.9	32.2	34.7	34.3
800	16.1	14.2	16.9	13.7	15.1	14.5	14.8	15.5
900	3.8	3.6	3.8	3.4	3.5	3.6	3.7	3.6

Table 9. Comparison of the HE and AHDE sequences across four basic heuristics for $M = 5$ and $D = 1500$.

	Webster		Jefferson		Bottleneck		Sequential	
n	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
100	1887.0	297.9	1859.4	309.9	1873.2	307.3	3253.4	2695.6
200	2249.2	244.2	2113.6	250.9	2258.1	253.0	1288.0	1442.4
300	2352.8	179.8	1638.7	183.7	1727.7	184.0	779.9	1236.6
400	3928.6	132.9	3027.4	135.2	3332.9	147.9	2643.1	2434.6
500	2621.2	93.8	1374.1	93.0	1591.6	102.7	875.1	1151.3
600	930.0	56.4	691.4	56.5	1140.5	55.0	466.6	467.1
700	370.8	24.6	718.2	26.4	396.8	30.4	179.7	218.9
800	603.4	17.5	274.4	16.1	225.5	15.1	78.8	68.7
900	174.9	8.3	202.5	7.6	737.0	17.9	37.9	22.5
1000	81.4	4.3	46.3	3.2	154.6	9.7	14.9	15.8
1100	12.0	1.4	12.6	0.8	17.5	3.1	10.0	4.8
1200	3.4	0.7	4.0	0.4	5.0	1.8	1.5	2.3
1300	0.3	0.1	0.1	0.0	0.2	0.5	0.7	0.7
1400	0.2	0.0	0.2	0.0	0.0	0.1	0.1	0.1

Table 10. Comparison of the HE and AHDE sequences across four basic heuristics for $M = 10$ and $D = 100$.

	Webster		Jefferson		Bottleneck		Sequential	
n	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
40	32.7	13.1	28.7	12.9	24.1	12.9	15.6	14.2
50	20.0	10.1	18.5	9.7	20.5	9.7	10.2	10.4
60	7.8	6.4	8.0	6.3	11.9	6.4	6.6	6.6
70	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5
80	1.5	1.5	1.5	1.4	1.4	1.4	1.5	1.5
90	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3

Table 11. Comparison of the HE and AHDE sequences across four basic heuristics for $M = 10$ and $D = 500$.

	Webster		Jefferson		Bottleneck		Sequential	
n	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
50	185.5	82.8	190.9	83.2	176.1	85.1	448.7	286.4
100	348.3	78.8	329.9	79.4	286.3	80.9	177.6	147.4
150	406.8	74.2	381.5	74.2	310.3	75.5	151.7	131.3
200	257.5	64.8	252.9	64.6	264.5	65.5	83.3	77.9
250	97.2	48.0	92.1	48.2	153.9	48.3	54.9	53.7
300	39.6	31.2	40.0	31.2	54.0	31.6	32.2	33.2
350	17.7	16.9	17.6	16.8	17.5	17.0	17.4	17.4
400	7.3	7.2	7.3	7.2	7.3	7.2	7.3	7.3
450	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5

Table 12. Comparison of the HE and AHDE sequences across four basic heuristics for $M = 10$ and $D = 1000$.

	Webster		Jefferson		Bottleneck		Sequential	
n	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
50	312.2	169.0	313.2	168.7	290.4	171.2	2832.8	1563.7
100	712.6	149.9	691.4	150.8	698.5	155.0	955.8	648.7
200	916.0	130.6	890.2	131.1	893.9	135.1	341.8	348.0
300	1228.8	96.2	1090.4	94.7	1077.1	97.6	489.2	360.0
400	475.4	85.0	463.3	84.9	616.4	87.2	172.2	139.1
500	196.0	68.9	255.6	69.0	334.8	69.9	108.3	82.7
600	78.5	51.1	109.4	50.8	258.4	51.2	64.6	57.7
700	32.3	29.9	30.8	30.1	32.1	30.8	35.7	34.8
800	16.5	13.9	16.4	13.7	15.2	14.6	14.9	14.4
900	3.8	3.5	3.8	3.4	3.5	3.5	3.5	3.7

Table 13. Comparison of the HE and AHDE sequences across four basic heuristics for $M = 10$ and $D = 1500$.

	Webster		Jefferson		Bottleneck		Sequential	
N	HE	AHDE	HE	AHDE	HE	AHDE	HE	AHDE
100	911.3	265.6	926.8	266.4	938.3	273.1	2752.0	1952.5
200	1696.3	235.2	1655.5	237.3	1654.2	241.3	931.8	795.2
300	1685.8	172.2	1669.0	173.3	1698.9	176.2	546.0	662.1
400	1597.5	134.4	1557.6	134.1	1733.5	137.7	863.0	655.9
500	2957.2	111.8	1337.6	111.8	1765.3	113.3	473.6	465.4
600	742.3	85.4	892.6	85.0	915.5	88.1	209.7	173.0
700	335.1	62.6	686.9	63.3	378.1	63.0	135.9	114.6
800	160.6	40.6	289.1	40.8	539.5	44.3	67.9	58.2
900	144.2	24.6	183.8	24.3	928.6	26.2	27.5	31.5
1000	78.7	14.2	52.1	13.6	120.3	15.0	14.9	16.4
1100	11.0	6.6	10.4	6.2	9.7	6.7	6.7	7.2
1200	6.0	2.6	7.2	2.6	3.9	2.7	2.8	2.8
1300	0.7	0.7	0.7	0.6	0.7	0.6	0.7	0.7
1400	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

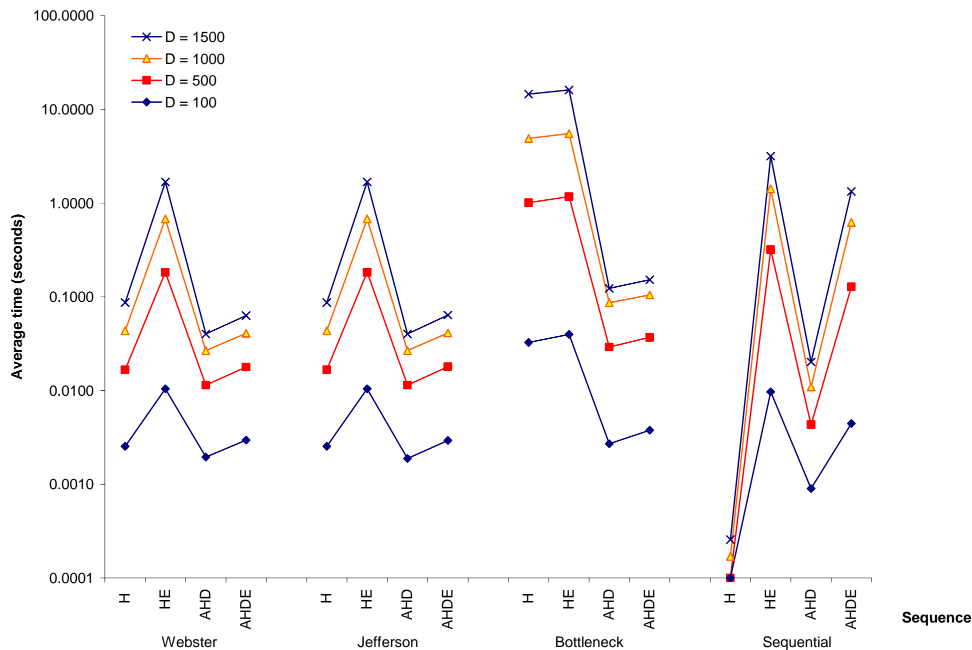

Figure 2. Average time required to generate different sequences using different heuristics when $M = 5$. Results are averaged over all instances with the given value of D. Note that the vertical axis is logarithmic to improve readability.

Table 14. Average Lower Bound.

		Lower Bound	
D	n	M = 5	M = 10
100	40	7.5	12.7
	50	6.5	9.5
	60	4.6	6.3
	70	2.8	3.5
	80	1.3	1.4
	90	0.3	0.3
500	50	72.9	81.6
	100	61.2	77.0
	150	44.0	72.9
	200	39.1	63.7
	250	31.2	46.9
	300	23.4	30.8
	350	14.1	16.7
	400	6.4	7.2
	450	1.5	1.5
1000	50	162.3	161.8
	100	149.8	144.8
	200	111.4	124.1
	300	86.2	88.5
	400	77.6	77.8
	500	63.9	63.9
	600	46.1	46.1
	700	28.2	28.2
	800	13.4	13.4
	900	3.4	3.4
1500	100	210.8	258.2
	200	169.6	225.5
	300	84.7	162.6
	400	29.9	126.8
	500	9.5	103.9
	600	2.9	79.3
	700	0.9	56.5
	800	0.2	37.3
	900	0.1	22.2
	1000	0.0	13.1
	1100	0.0	6.0
	1200	0.0	2.5
	1300	0.0	0.6
	1400	0.0	0.1

7 Summary and Conclusions

This paper presents an aggregation approach for the problem of minimizing RTV when there are multiple servers to process the products. We combined this approach with various heuristics for the multiple-server RTV problem in order to determine when aggregation is useful.

The results show that, in most cases, combining aggregation with other heuristics does dramatically reduce both RTV and computation time compared to using the heuristics without

aggregation (when the exchange heuristic is used to improve the solutions found). In these cases, the solutions generated by the heuristics have large values of RTV, and aggregation provides a way to find much better solutions. Aggregation is particularly effective in combination with the Webster, Jefferson, and bottleneck heuristics. Comparing the RTV of the sequences generated using aggregation to a lower bound shows that the sequences are near-optimal solutions.

We can see why aggregation improves RTV if we consider a set of products with the same demand. The Webster, Jefferson, and bottleneck heuristics tend to put copies of these products next to each other, which increases the RTV of products with higher demand. Because it combines them into one group, aggregation spreads out the products with the same demand, leaving space for the products with higher demand.

Due to the larger computational effort of the bottleneck heuristic, using aggregation with the Webster and Jefferson heuristics (instances of the parameterized stride scheduling heuristic) will be the most effective approach. We note that Corominas et al. [11] found that, without aggregation, the Webster and Jefferson heuristics generated single-server sequences with higher RTV than those generated by the bottleneck heuristic.

On the other hand, when the number of products is near the total demand (and each product has very small demand), combining aggregation with other heuristics does not reduce RTV compared to using the heuristics without aggregation (when the exchange heuristic is used to improve the solutions found). Still, in these cases, aggregation does reduce the computational effort needed to generate sequences.

The aggregation procedure described in this paper is simple but effective. However, it is possible to create more sophisticated aggregations that more intelligently combine products in order to minimize the number of products in the highest level of aggregation, with the goal of aggregating the products into one group for each server. If there is only one group for each server, the disaggregation leads directly to a zero RTV sequence. It will be useful to consider the use of optimization approaches (which can solve only small RTV problems) on the aggregated instances to determine if aggregation can reduce RTV and computational effort with that approach as well.

Acknowledgements This work was motivated by a collaboration with the University of Maryland Medical Center. The author appreciates the help of Leonard Taylor, who introduced the problem, provided useful data, and recognized the value of the work presented here.

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. 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.
15. Herrmann, Jeffrey W., Using Aggregation to Reduce Response Time Variability in Cyclic Fair Sequences, Technical Report 2008-20, Institute for Systems Research, University of Maryland, August, 2008.
16. 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.
17. 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.
18. Steiner, George, and Scott Yeomans, Level Schedules for Mixed-Model, Just-in-Time Processes, Management Science, 39, 728-735, 1993.