OPERATIONS RESEARCH
Vol. 53, No. 4, July–August 2005, pp. 675–688
ISSN 0030-364X | EISSN 1526-5463 | 05 | 5304 | 0675

informs®
DOI 10.1287/opre.1040.0201
© 2005 INFORMS

Recovering Cyclic Schedules Using Dynamic Produce-Up-To Policies

Donald D. Eisenstein
Graduate School of Business, The University of Chicago, Chicago, Illinois 60637, don.eisenstein@chicagogsb.edu

We consider facilities that follow a cyclic schedule to replenish the inventory of a set of items through production by a shared resource. We introduce a dynamic produce-up-to policy that recovers the target cyclic schedule after a single disruption, and is also shown to be effective when disruptions are more frequent. Our policy is more flexible than traditional recovery policies in that our policy is able to adjust the amount of idle time observed during recovery in response to disruptions, and yet re-establish the target idle time as the schedule recovers. This results in a policy that not only saves cost and time, but provides better schedule stability than other recovery procedures. Furthermore, unlike simple produce-up-to policies, our policy is anticipatory—replenishments will be speeded up or delayed, whichever is necessary, to help avoid congestion at the shared resource. In addition, recovery is controlled by a single "knob" or parameter that can tune recovery to be fast and aggressive (frequent setups and small batches) or slow and methodical (few setups and larger batches). Finally, our policy is easy to implement, augmenting a traditional produce-up-to policy with a simple set of counters that control replenishment decisions.

Subject classifications: production/scheduling: recovering cyclic schedules, bucket brigades; mathematics: discrete dynamical systems.
Area of review: Manufacturing, Service, and Supply Chain Operations.
History: Received June 2000; revisions received June 2001, July 2002, July 2004; accepted July 2004.

Introduction

There is considerable work in the literature on determining good target schedules for cyclic production systems, commonly known as economic lot-scheduling problems. Here m items, each of which experiences constant demand, are replenished in a cyclic fashion on a shared machine. Production occurs at a constant rate (after a possible deterministic setup time). The target cyclic schedule can be described by three n-dimensional vectors, $(\mathbf{f}, \mathbf{u}, \mathbf{t})$: A vector of item indices $\mathbf{f}$, indicating the sequence of items to produce in each cycle; a vector of idle times $\mathbf{u}$; and a vector of production run times $\mathbf{t}$. The target schedule behaves as follows: The system is first idled for time u_1, then the item in the first position is both set up and produced in time t_1, then the system is idled for u_2, and then set up and produced for t_2, and so forth until the item in the nth position is produced. In general, $n \geqslant m$ because some items may be repeated in the schedule. Target schedules are sought that minimize inventory and production costs (see, for instance, Delporte and Thomas 1977, Dobson 1987, Maxwell 1964, Zipkin 1991).

We assume that a target cyclic schedule is given. We develop a new procedure that recovers the target cyclic schedule after a single shock, such as when demand spikes or a machine fails. However, our procedure is robust—it can be used to respond to general perturbations as in the stochastic economic lot-scheduling problem (SELSP)—see Sox et al. (1999) for a nice overview of this literature.

Our procedure can be thought of as lying within a spectrum of two other recovery procedures—one fixes idle times during recovery (a policy we call **u**Fixed), while the other ignores idle time (a policy we call **u**Ignore).

Optimal cyclic schedules for the economic lot-scheduling problem may have planned idle time, but in addition systems are commonly designed with idle time to hedge against uncertainties (Bourland and Yano 1994, provide a nice overview of the use of planned idle time to help absorb uncertainties such as variances in demand, manpower, and equipment). We consider three criteria when evaluating the effectiveness of a recovery procedure: (1) cost, (2) relative stability, and (3) absolute stability.

We measure the *cost* of a cyclic schedule in the traditional manner—holding, setup, and back-order costs. How a system reacts to a shock(s) will affect these costs.

We define *relative stability* as the ability to preserve the target idle time u_i for each item at each iteration during recovery. Such a recovery procedure that concentrates on relative stability (such as a **u**Fixed policy) will tend to allow the schedule to drift (for example, an item previously produced on Wednesday is now produced on Thursday). This is because by keeping the u_i stable, shocks must be absorbed by allowing the production time to deviate from the target time t_i. Maintaining target idle times provides some managerial advantages: See Schonberger (1982) for a discussion of the importance of maintaining daily slack capacity to help absorb variances in manpower and

equipment as well as maintaining high quality. Schonberger (1982) and Suzaki (1987) discuss the importance of daily machine checks.

We define *absolute stability* as the ability to preserve the cycle time $u_i + t_i$ for each item at each iteration during recovery—so that the recovery of one item does not disrupt the cycle of the next item. Thus, such a recovery procedure that concentrates on absolute stability will tend to prevent schedule drift (so, for example, maintaining that a particular item is always produced on Wednesdays). However, the reduction in drift is typically accomplished at the expense of relative stability; that is, to preserve the cycle time of an item the idle time might be shortened or lengthened during recovery to accommodate a shock in the system. Absolute stability is important to coordinate and schedule raw material deliveries into the process and schedule deliveries of product out of the process. It also helps to better schedule manpower needs, which might change depending on the item(s) produced. While it is impossible, given a large enough shock, for any recovery procedure to maintain absolute stability (there simply may not be enough available idle time to absorb the shock in demand), it is advantageous for a procedure to remain "close," so that management can recover absolute stability with small amounts of rescheduling at the control level (such as with overtime or outsourcing). See Bourland and Yano (1994) for a discussion of the importance of absolute stability and procedures at the control level to maintain it. As we will later detail, our procedure will guarantee absolute stability when the shock is not too large and the parameters to our policy are set correctly.

Consider Allied Foods in Atlanta, Georgia, which produces in a cyclic fashion about 20 different formulas of canned pet food. One line has considerable planned idle time of over 20%, which is distributed throughout the target schedule. Maintaining absolute stability is critical for Allied for two reasons: first, so that personnel can be reliably scheduled to work at other tasks when the line is idle, and second, to plan the delivery of raw materials that change depending on the item being produced.

Gallego (1990) examines a class of **u**Fixed linear control policies to recover a target schedule after a single shock. At the start of the production cycle, the difference between the current inventory levels and the target is multiplied by a linear transformation or gain matrix, resulting in a production plan for the next cycle. Only the production run times are control variables, because the idle-time vector is fixed. Gallego solves a control problem that determines the gain matrix that minimizes the excess over average cost of recovery.

Gallego (1994) extends this work by examining a special case, a **u**Fixed-PUT (**u** Fixed Produce-Up-To) policy. As each item is up for replenishment, production continues until an up-to point is reached. Gallego shows that a **u**Fixed-PUT policy is a linear control policy—that is, there is a gain matrix that results in the policy. He also shows that a **u**Fixed-PUT policy solves the control problem whenever a

Proportionality Condition

$$\hat{\pi}_i p_i = \theta \quad \text{for each item } i = 1, \ldots, m, \tag{1}$$

holds; where $\hat{\pi}_i$ is the backorder cost, p_i is the production rate, and the demand rate for each item is normalized to one.

Produce-up-to policies afford some advantages: No gain matrix calculation is required, and the system is able to react and adjust as each item is up for production.

At the other end of the spectrum is a **u** Ignore Produce-Up-To policy. A **u**Ignore-PUT policy simply queues an item for replenishment whenever its target reorder point is reached. We show that such a policy will recover a schedule $(\mathbf{f}, \mathbf{u}', \mathbf{t})$, where $\mathbf{u}'$ is some vector of idle times that redistributes the available idle time in some other unspecified and unconstrained way across the schedule.

We develop a new class of policies we call *Dynamic-PUT* (Dynamic Produce-Up-To) policies. Unlike these other policies, a Dynamic-PUT policy is able to reduce or eliminate available idle time in the target schedule during recovery when the system is backlogged, or insert additional idle time during recovery if the system is overstocked, but then gradually restore the idle times to their target levels—and provide this flexibility while still maintaining the simplicity and advantages of a produce-up-to policy.

We show that a Dynamic-PUT policy will recover faster and with less cost than an optimal **u**Fixed policy whenever the proportionality condition (1) holds and there is some idle time available in the target schedule. Simulations show that the cost savings can be substantial (for instance, we find savings of 15%–20% with only 5% idle time available in the target schedule). Furthermore, even when the proportionality condition does not hold, simulations indicate that our policy performs better than the optimal **u**Fixed policy whenever idle time in the target schedule exceeds just 1%.

We also explore more pervasive variance by simulating a system that is subjected to multiple shocks over time. While a **u**Fixed policy by definition maintains relative stability, we find that a Dynamic-PUT policy performs better in terms of cost and absolute stability. A **u**Ignore policy does not restore the idle time in any predictable way, and thus will quickly lose the ability to maintain either relative or absolute stability. We even find that this lack of stability can cause a **u**Ignore policy to perform worse than a Dynamic-PUT policy in terms of cost, but overall the cost of a **u**Ignore and Dynamic-PUT policy are very close. Thus, in general, a Dynamic-PUT policy performs very well in terms of cost and both relative and absolute stability.

A Dynamic-PUT policy, if desired, allows items in severe backlog to easily and naturally move up in the production sequence. In addition, it allows a manager to control the procedure with a single "knob" that directs the

convergence back to the target to be quick and aggressive (frequent setups and small batches to emphasize stability), or slow and methodical (few setups and larger batches).

Furthermore, the policy is easy to implement. Whenever an item is replenished, its inventory is always restored to a static up-to point. However, in addition we also restore a set of counters or tokens for each item. The tokens for each item are depleted in a simple way along with its inventory, and when the tokens for an item reach zero, the item queues for replenishment.

We are able to derive simple conditions for the policy to recover the target schedule after any shock. Furthermore, we are able to determine conditions when recovery under a Dynamic-PUT policy will maintain absolute stability perfectly and recover the target schedule within one production cycle.

1. Cyclic Scheduling

We let T, the cycle length, be the total elapsed time for a target schedule to complete the entire cycle, after which the schedule repeats. For notational simplicity we assume a rotation schedule, $n = m$ (no items repeat in the schedule).

We assume that all demand is backordered. We normalize the demand rate of each item to one, so that each item experiences T units of demand continuously and smoothly over the target cycle length. The production rate p_i is scaled accordingly so that the production time t_i is given by

$$t_i = s_i + \frac{T}{p_i}, \tag{2}$$

where s_i is the setup time required for item i.

Without loss of generality, we assume that in the target schedule the inventory level of each item is zero just as setup and production begins (we expect that an optimal target schedule shifts our inventory cycle by some amount up or down). A cyclic schedule $(\mathbf{f}, \mathbf{u}, \mathbf{t})$ can also be represented as an equivalent schedule $(\mathbf{f}, \mathbf{u}, \mathbf{B})$, where $\mathbf{B}$ is a vector of base stock or up-to points. Thus, at the target schedule, when the inventory of each item hits zero, it is set up for production and then produced until its inventory is built up to level B_i (see Figure 1). The corresponding up-to point for item i is

$$B_i = \frac{(p_i - 1)T}{p_i} - s_i. \tag{3}$$

Thus, by combining Equations (2) and (3),

$$t_i = \frac{B_i + p_i s_i}{p_i - 1}.$$

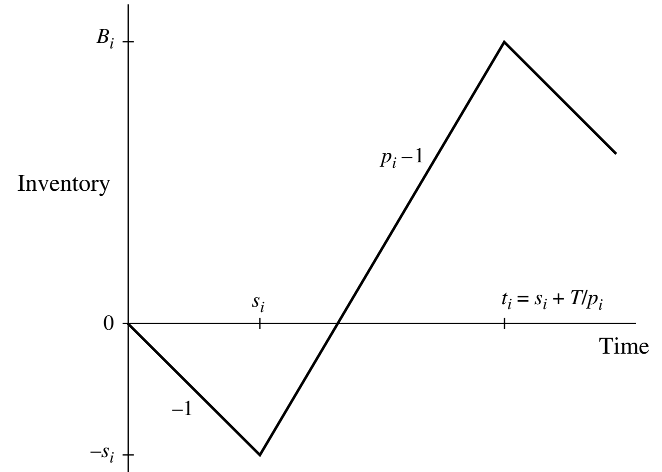
We define the target *cycle time*, τ_i, for item i to be the target time to idle, set up, and produce the item,

$$\tau_i = u_i + t_i, \tag{4}$$

thus from Equations (3) then (2) and (4)

$$B_i = \sum_{q \neq i} \tau_q + u_i \tag{5}$$

Figure 1. The target processing time $t_i = s_i + T/p_i$ corresponds to an up-to point $B_i = (p_i - 1)T/p_i - s_i$.



and

$$T = \sum_{i=1}^{n} \tau_i = \sum_{i=1}^{n} \frac{T}{p_i} + \sum_{i=1}^{n} s_i + \sum_{i=1}^{n} u_i. \tag{6}$$

We assume that there is some slack capacity in the system (idle and/or setup time), otherwise, recovering from any general shock would be impossible. From Equation (6), we have the following condition for the system to have slack capacity.

Assumption 1 (Slack Capacity).

$$\sum_{i=1}^{n} \frac{1}{p_i} < 1.$$

2. A Dynamic Produce-Up-To Policy

Our policy is a *produce-up-to* policy, so that whenever an item begins a replenishment, we always produce enough to move its inventory level up to B_i units.

Our policy introduces a set of counters or *tokens* for each item that are used to trigger replenishments. As the inventory level of item i is replenished to B_i, we will restore its token level up to a static value we denote as Z_i. We introduce a velocity parameter for each item, $v_i \geqslant 1$. We set

$$Z_i = \sum_{q \neq i} \tau_q + u_i v_i \quad \text{or} \tag{7}$$

$$= B_i + u_i(v_i - 1) \quad \text{from Equation (5).} \tag{8}$$

Dynamic-PUT Policy. Produce items according to the target sequence. For each item, select parameter v_i and calculate an inventory up-to point B_i from Equation (3), and a token up-to point Z_i from Equation (8). Whenever an item, *except the next in sequence for production*, experiences a unit of demand, decrement its inventory by one and its token level by one. Whenever the *next* item in sequence

for production experiences a unit of demand, decrement its inventory by one and its token level by its velocity parameter v_i. Whenever the machine is idle resume production of the next item in sequence when its token level is zero or less. When producing item i, produce to bring its inventory level up to B_i and restore its token level to Z_i.

Tokens are used to guide how long the system should idle between production runs. The tokens provide a manager with a natural and familiar threshold policy that corresponds to a simple inventory policy. Tokens also provide a manager with a natural extension to our policy that allows any item whose tokens first reach zero to begin production—thus providing a simple mechanism to move items in severe deficit up in the production sequence, and items whose demand has waned to move back in the production sequence.

2.1. System Dynamics

We now examine the behavior of a Dynamic-PUT policy *after a single shock*. We can think of a shock as a perturbation to the inventory levels. Our analysis assumes that after the shock, the system behaves deterministically and that demand is continuous. We restrict our attention to the state of the system just after production of each item is completed. A new *iteration* begins as each item completes a production run, and thus it requires n iterations to produce all items. After each production run (at the start of each iteration) we index the new position of each item—so that $[1]$ indexes the label of the item just replenished and so on until $[n]$ indexes the label of the item next to be replenished. We let $x_{[j]}^{(k)}$ be the inventory level and $y_{[j]}^{(k)}$ be the token level of the item in position j at the start of iteration k, just after the $k-1$st production. Thus, $x_{[n]}^{(k)}$ and $y_{[n]}^{(k)}$ give the inventory and token level of the item next up for production at the start of iteration k. We study the trajectory $\{(\mathbf{x}^{(0)}, \mathbf{y}^{(0)}), (\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(k)}, \mathbf{y}^{(k)}), \ldots\}$ of the dynamical system.

We also write the up-to points for the item at position j at the start of iteration k as $B_{[j]}^{(k)}$ and $Z_{[j]}^{(k)}$; its static parameters as $s_{[j]}^{(k)}$, $p_{[j]}^{(k)}$, and $v_{[j]}^{(k)}$; and its target schedule parameters as $u_{[j]}^{(k)}$, $t_{[j]}^{(k)}$, and $\tau_{[j]}^{(k)}$.

Item $[n]$ at the start of iteration k has token level $y_{[n]}^{(k)}$, so we can write the realized idle time as

$$u^{(k)} = \frac{(y_{[n]}^{(k)})^+}{v_{[n]}^{(k)}}.$$

It is then necessary to increase its stock by $B_{[n]}^{(k)} - x_{[n]}^{(k)} + u^{(k)}$ units, so iteration k will realize the cycle time

$$\begin{aligned} \tau^{(k)} &= u^{(k)} + s_{[n]}^{(k)} + \frac{B_{[n]}^{(k)} - x_{[n]}^{(k)} + u^{(k)} + s_{[n]}^{(k)}}{p_{[n]}^{(k)} - 1} \\ &= u^{(k)} + \frac{B_{[n]}^{(k)} - x_{[n]}^{(k)} + u^{(k)} + p_{[n]}^{(k)} s_{[n]}^{(k)}}{p_{[n]}^{(k)} - 1}. \end{aligned} \tag{9}$$

The following dynamics equations map a Dynamic-PUT policy from one iteration (production run) to the next:

Dynamic-PUT

$$\begin{aligned} x_{[1]}^{(k+1)} &= B_{[n]}^{(k)}, \\ x_{[j]}^{(k+1)} &= x_{[j-1]}^{(k)} - \tau^{(k)}, \quad j = 2, \ldots, n, \\ y_{[1]}^{(k+1)} &= Z_{[n]}^{(k)}, \\ y_{[j]}^{(k+1)} &= y_{[j-1]}^{(k)} - \tau^{(k)}, \quad j = 2, \ldots, n. \end{aligned} \tag{10}$$

It is notationally convenient to assume that at the start of iteration 0 (after a shock) the inventory level for each item i begins no greater than its up-to point—this assumption is reasonable, because a shock can never coerce one to produce beyond the up-to point of an item. We also note that the token levels at iteration 0 are set according to Equation (8). Therefore, we have the following by assumption at iteration 0 and by Equations (10) for all subsequent iterations:

$$B_{[j]}^{(k)} - x_{[j]}^{(k)} = Z_{[j]}^{(k)} - y_{[j]}^{(k)} \geqslant 0 \quad \text{for each } j = 1, \ldots, n \text{ and all } k \geqslant 0. \tag{11}$$

Equations (10) are stable only at the target schedule.

THEOREM 1. *Given a target schedule* $(\mathbf{f}, \mathbf{u}, \mathbf{B})$, *along with parameters* p_i *and* s_i, *the Dynamic-PUT system given by Equations* (10) *is stable only at the target schedule.*

PROOF. After expanding Equations (10) for n iterations, there is a unique feasible solution. ☐

From Equations (8) and (11) we can also write the realized idle time for iteration k as

$$u^{(k)} = \left(\left(1 - 1/v_{[n]}^{(k)}\right) u_{[n]}^{(k)} + \left(1/v_{[n]}^{(k)}\right) x_{[n]}^{(k)}\right)^+. \tag{12}$$

Thus, our Dynamic-PUT policy can be viewed as setting the idle time as a convex combination of the target idle time and the expected stockout time. As $v \to \infty$, our policy tends to a **u**Fixed-PUT policy, and when $v = 1$, our policy is a **u**Ignore-PUT policy. Thus, a manager could implement our policy by setting the predicted idle time according to Equation (12) at the start of an iteration, and then decrementing the remaining idle time as each unit is demanded.

We now look for policies that make the target schedule an attractor. The system will self-balance or attract to the target schedule if after any shock that results in inventory position $\mathbf{x}^{(0)}$, the trajectory $\{\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(k)}, \ldots\}$ converges. We will find that we can make the target schedule an attractor if the velocity parameters are set correctly.

2.2. Self-Balance

If the velocity parameter v_i for each item is set to a value of at least p_i, then the system will self-balance—the target schedule will become an attractor.

THEOREM 2. *If $v_i \geqslant p_i$ for each item i, then the Dynamic-PUT system converges to the target schedule* $(\mathbf{f}, \mathbf{u}, \mathbf{B})$.

PROOF. See the appendix, §A.1. □

2.2.1. Quick Self-Balance. If each v_i is set to p_i, then a Dynamic-PUT policy can recover the target schedule after a single shock very quickly (within the cycle length T) if the shock is not too large. Furthermore, absolute stability will be maintained during recovery. This occurs because the decrease (increase) in idle time due to a shock when $v_i = p_i$ exactly matches the increase (decrease) in production time.

THEOREM 3. *If $v_i = p_i$ for each i, and if at some iteration k, $x_{[j]}^{(k)} > \hat{x}_{[j]}^{(k)} - u_{[j]}^{(k)} v_{[j]}^{(k)}$ for each j, then $\tau^{(q)} = \tau_{[n]}^{(q)}$ for all $q \geqslant k$ and $x_{[j]}^{(q)} = \hat{x}_{[j]}^{(q)}$ for all j and for all $q \geqslant k+n$; where we let $\hat{x}_{[j]}^{(k)}$ denote the target inventory level for the item at position j at the start of iteration k.*

PROOF. See the appendix, §A.2. □

Therefore, when recovering from a surplus or limited deficit, with each $v_i = p_i$ the system maintains perfect absolute stability as each recovery cycle time matches the target cycle time, and thus the system recovers the target schedule after a single production of each item. This is in contrast to **u**Fixed or **u**Ignore policies that will lose absolute stability with the smallest shock.

2.3. An Example

We now consider an example of a three-item cyclic schedule. To simplify the exposition without sacrificing the main ideas, the items have identical production parameters.

EXAMPLE 1. Let $n = 3$, a setup time of $s = 0$, a target cyclic schedule described by $\mathbf{u} = (25, 25, 25)$ and $\mathbf{t} = (75, 75, 75)$, and thus the target cycle time for each item is $\tau_i = 100$ and target cycle length $T = 300$. By Equation (2), we have a production rate of $p_i = 4$ for each item, and by Equation (3), a produce-up-to point for each item of $B_i = (4-1)300/4 = 225$. Suppose that we choose $v_i = 4$ for each item, then by Equation (7) we have each $Z_i = 100 + 100 + 25(4) = 300$.

Because the three items are identical, the target schedule is characterized by the single inventory vector $\hat{\mathbf{x}} = (225, 125, 25)$ after each replenishment, and from Equation (11), we have $\hat{\mathbf{y}} = (300, 200, 100)$.

Suppose that we begin our system at the target so that $\mathbf{x}^{(0)} = \hat{\mathbf{x}}$ and $\mathbf{y}^{(0)} = \hat{\mathbf{y}}$. During iteration 0 the system will idle the target time of $25 = 100/4$ until the tokens of the item in the third position reach 0. Then, the inventory of the third item will be replenished in an additional 75 time units, at which point its inventory is replenished back to 225 and its tokens to 300. Furthermore, the inventory and token levels of the other two items have each decreased by 100 units—the system has returned to its target. Thus, when the system is at its target point with no perturbations to the system, the tokens play no active role—as the token level of each item reaches zero, so does its inventory level.

Table 1. The out-of-balance system is quickly restored using the Dynamic-PUT policy.

k	$x_{[1]}^{(k)}$	$x_{[2]}^{(k)}$	$x_{[3]}^{(k)}$	$y_{[1]}^{(k)}$	$y_{[2]}^{(k)}$	$y_{[3]}^{(k)}$	$u^{(k)}$	$t^{(k)}$	$\tau^{(k)}$
0	225	160.0	105.0	300	235.0	180.0	45.0	55.0	100.0
1	225	125.0	60.0	300	200.0	135.0	33.75	66.25	100.0
2	225	125.0	25.0	300	200.0	100.0	25.0	75.0	100.0

Note. The inventory levels, token levels, idle time, production time, and cycle time are shown for each iteration.

However, now suppose that our system has experienced a shock(s) leaving the system just after a replenishment with, for example, inventory vector $\mathbf{x}^{(0)} = (225, 160, 105)$ (we reset the iteration counter to 0). The corresponding token vector is $\mathbf{y}^{(0)} = (300, 235, 180)$ by Equation (11). Table 1 shows the progression of iterations as balance is quickly restored to this system. After only three iterations (a single production of each item) the system is restored to balance. This example of recovery illustrates Theorem 3—absolute stability is maintained during recovery (each cycle time is 100), and the target is recovered after a single production of each item.

In general, the self-balancing of the system is accomplished by utilizing token space to add a small amount of coordination between the inventory stocks—each item must keep track of its sequence so its token velocity can be set accordingly. Simulations lead us to conjecture that this coordination will also lead to self-balance for all $1 < v_i < p_i$, but v_i in this range complicates the dynamics considerably. However, when each $v_i = 1$, this coordination is eliminated and the system fails to converge to the target schedule.

2.4. Imbalance: A uIgnore-PUT Policy

When each $v_i = 1$ the tokens never play an active role, because token levels are always the same as inventory levels. In this case, our policy degenerates into a **u**Ignore-PUT policy, where an item queues for replenishment whenever its inventory reaches zero. In this case, the system fails to converge to the target schedule—**f** and **t** are recovered, but the idle-time vector **u** is not. Consider the following example with the same system used in Example 1.

EXAMPLE 2. Let $n = 3$, $\mathbf{B} = (225, 225, 225)$, $\mathbf{u} = (25, 25, 25)$, each $v_i = 1$, $s_i = 0$, $p_i = 4$, $t_i = 75$, and $T = 300$. If the system begins with $\mathbf{x}^{(0)} = (225, 75, 0)$, then the system will cycle through the three points $\mathbf{x}^{(3q)} = (225, 75, 0)$, $\mathbf{x}^{(3q+1)} = (225, 150, 0)$, and $\mathbf{x}^{(3q+2)} = (225, 150, 75)$ for $q = 0, 1, 2, \ldots$. The idle times in the system are not evenly distributed as in the target schedule—stability is lost. The system produces all three items up to 225 sequentially and then idles for 75 time units before repeating the cycle. If the system is subjected to a shock, then the system will converge to a new three-cycle. There is an infinite number of

such three-cycles, each dividing up the 75 units of idle time in a different way.

The following theorem establishes that when each $v_i = 1$, our policy degenerates to a **u**Ignore-PUT policy that will converge to some n-cycle.

THEOREM 4. *If each $v_i = 1$, then the Dynamic-PUT system is a **u**Ignore-PUT policy, and it converges to a d-cycle for some $d \leqslant n$.*

PROOF. See the appendix, §A.3. □

We note that the cyclic schedule resulting from a **u**Ignore-PUT policy will exhibit the same costs as the target—how idle time is distributed throughout a cyclic schedule does not affect its cost. However, under such a policy stability is lost. We will later see that such a loss of stability can also cause the policy to accumulate extra costs when a system experiences frequent shocks.

3. Recovery Performance

We examine the recovery of a target schedule $(\mathbf{f}, \mathbf{u}, \mathbf{t})$ after a single shock leaves the system in an overall deficit and then an overall surplus. Then, we examine when a system is subjected to multiple shocks over time. Here the target may never fully recover, and thus not only is cost an issue, but how well a policy maintains relative and absolute stability. (Proofs in this section are located at http://or.pubs.informs.org/Pages/collect.html.)

3.1. Deficit Recovery from a Single Shock

We first examine recovery after a shock leaves the system in a deficit. We prove that Dynamic-PUT recovery is superior to **u**Fixed-PUT recovery in cost and speed if there is any idle time available in the target schedule and the proportionality condition (1) holds. Simulations indicate that the savings can be substantial. (If there is no idle time in the target schedule, then we show that the recovery procedures are identical.)

We then examine schedules for which the proportionality condition does not hold by comparing our recovery performance to the general optimal **u**Fixed policy. Simulations indicate that our policy is superior as long as there is a small amount of idle time available in the target schedule.

3.1.1. Dynamic-PUT Versus uFixed-PUT Recovery.

Gallego (1994) studies **u**Fixed-PUT policies.

uFixed-PUT Policy. Produce items according to the target sequence. First, idle for time u_1, set up for time s_1, then produce Item 1 up to B_1; then idle for time u_2, set up for time s_2, then produce Item 2 up to B_2, and so on until the last item is produced up to B_n. Repeat.

When the system begins in a deficit and the proportionality condition holds, **u**Fixed-PUT recovery minimizes the excess over average cost of recovery (subject to idle times being fixed).

One way to view recovery using a Dynamic-PUT policy is that it effectively increases the production rate when the system is in a deficit by reducing or eliminating idle time. Thus, at first, the effective production rate is increased the most, until gradually it decreases as the system regains balance.

The following lemma details this distinguishing feature of Dynamic-PUT recovery.

LEMMA 1. *If at some iteration the next item up for production is in a deficit, and the target schedule has some available idle time, then a Dynamic-PUT policy will idle an amount strictly less than the target idle time.*

We can now establish the cost and speed advantage of recovery.

THEOREM 5. *Consider a system that begins with each item in a deficit and that has some idle time in the target schedule. Then, a Dynamic-PUT policy with each $v_i \geqslant p_i$ will recover faster and with less cost than a **u**Fixed-PUT policy.*

Our proof also establishes the following.

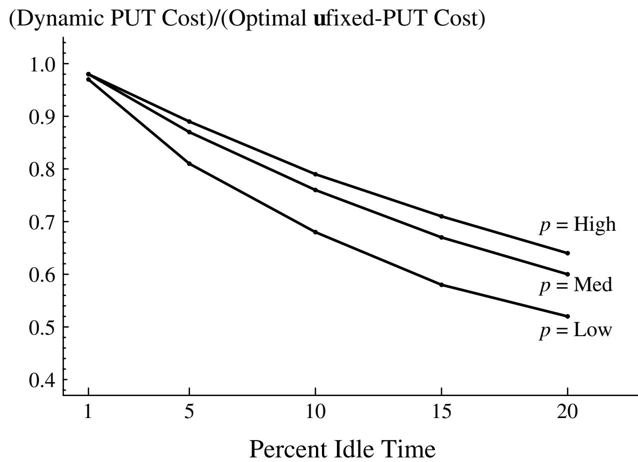
COROLLARY 1. *Consider a system that begins with each item in a deficit and that has no idle time in the target schedule. Then, recovery for a Dynamic-PUT policy with each $v_i \geqslant p_i$ will be identical to a **u**Fixed-PUT policy.*

We have simulated the **u**Fixed-PUT and Dynamic-PUT policies to investigate the savings available. We generated instances, each with identical items, as detailed in Table 2. For each run we calculated the excess over average cost of the optimal cyclic schedule for both policies. For each n we generated an instance for each possible combination of the parameters. Runs were started with inventory beginning at

Table 2. Set of instances generated to compare Dynamic-PUT recovery with **u**Fixed-PUT recovery.

	$n = 5$	$n = 10$	$n = 20$
Production rate	{6, 11, 16}	{15, 20, 25}	{25, 35, 45}
Backorder costs	{20, 200}	{20, 200}	{20, 200}
Holding costs	{1, 10}	{1, 10}	{1, 10}
Setup costs	{50, 500}	{50, 500}	{50, 500}
Starting inventory	{ZERO, NEG}	{ZERO, NEG}	{ZERO, NEG}
Percentage of idle time	{1, 5, 10, 15, 20}	{1, 5, 10, 15, 20}	{1, 5, 10, 15, 20}

Figure 2. When idle times are small, the cost of the Dynamic-PUT policy is close to that of the **u**Fixed-PUT policy, but when the idle time in the target schedule increases, the savings is significant.



Note. We plot the average over a range of cost data for (L)ow, (M)edium, and (H)igh production rates.

either 0 (ZERO) or $-B$ (NEG). Therefore, for each value of n, 240 instances were generated. For each instance we calculated the optimal **u**Fixed cost (using the **u**Fixed-PUT policy because the proportionality condition holds), and the cost of a Dynamic-PUT policy with a number of different v values (usually five)—over 3,000 runs were made. Setup times were set implicitly for each run to satisfy the optimal cycle time T, the production rate p, and the percent idle time for each run.

The relative performance of Dynamic-PUT and **u**Fixed-PUT were very similar for each value of n, and for either inventory start position. Figure 2 summarizes the results for 10 items, each starting at zero inventory. The average cost of Dynamic-PUT recovery divided by the cost of **u**Fixed recovery is plotted for each production rate value ((L)ow, (M)edium, and (H)igh) and idle percentage over all other combinations of parameters. For a given idle percentage and production rate, the performance ratio had very little variance (the coefficient of variation was always <0.10), so we plot the average of all runs. Our simulations indicate that, as we would expect, Dynamic-PUT recovery was more and more advantageous as the amount of idle time in the target schedule increased. For instance, when the percentage of idle time in the target schedule was 20%, and the production rate its lowest ($p = 15$), the cost of Dynamic-PUT recovery averaged just over half of **u**Fixed recovery. However, when the idle percentage was only 1%, the cost savings averaged about 4%. In addition, Dynamic-PUT recovery was more advantageous when the production rate was lowest. This is because with a low production rate, recovery is especially difficult, and thus the ability to utilize any available idle time is all the more valuable.

Table 3. Set of instances generated to compare Dynamic-PUT recovery with **u**Fixed recovery.

	A	B	C	D
Production rate	[15, 15]	[12.2, 18.2]	[11.0, 20.0]	[9.8, 22.0]
Backorder costs	[60, 60]	[34, 94]	[25, 117]	[19, 147]
Holding costs	[1, 10]	[1, 10]	[1, 10]	[1, 10]
Setup costs	[50, 500]	[50, 500]	[50, 500]	[50, 500]

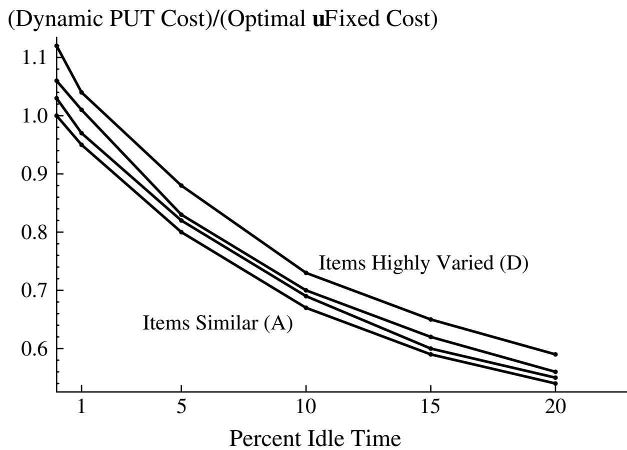
3.1.2. Dynamic-PUT Versus uFixed Recovery. We now examine recovery when the proportionality condition (1) does not hold. In this case, the optimal **u**Fixed policy is not a produce-up-to policy, but a more general strategy. Thus, Dynamic-PUT is constrained by a produce-up-to strategy, whereas **u**Fixed recovery is constrained by fixing idle times during recovery—we would not expect either to dominate in all instances.

We ran simulations to compare Dynamic-PUT recovery with the optimal **u**Fixed recovery (MATLAB, from The Mathworks, Inc., was used to solve the Algebraic Ricatti Equation necessary to generate the optimal gain matrix for the **u**Fixed policy). We generate instances of optimal rotation schedules. Each run was made with 10 items each starting from a small negative inventory position. For each of the groups (A,B,C,D) shown in Table 3 and for each target idle percentage in $\{0, 1, 5, 10, 15, 20\}$, we generated 30 random problems varying the parameters uniformly over the ranges shown in the table. Setup times were set implicitly for each run to satisfy the optimal cycle time, production rates, and the percent idle time. Thus, runs in Column A generated items with the same production rate and backorder costs, and so the proportionality condition holds, and thus a Dynamic-PUT policy will perform no worse than a **u**Fixed policy (and better if there is any idle time). However, as we move to Columns B, C, and finally D, the proportionality condition fails to hold with more and more intensity, and thus we expect the constraint of a produce-up-to policy to become more costly.

For each run we calculate the average cost of Dynamic-PUT recovery divided by the cost of **u**Fixed recovery. Figure 3 summarizes the results, where each data point is the average over the 30 runs (the coefficient of variation was very small between these runs, always less than 0.03). If there was no idle time in the target schedule, then **u**Fixed recovery outperformed Dynamic-PUT recovery by as much as 12% when the items varied widely. However, when the idle time increased to just 1% the policies were similar—the **u**Fixed policy had a small advantage when the items were the most varied (Groups C and D) and Dynamic-PUT had a small advantage when items were most similar (Groups A and B). When idle times exceeded 1%, the Dynamic-PUT policy was superior. With just 5% idle time, the Dynamic-PUT policy saved between 10% and 20% of the cost of a **u**Fixed policy.

Thus, our results indicate that a manager should use a Dynamic-PUT policy even with a wide dispersion of item

Figure 3. When idle times are very small, the cost of the optimal **u**Fixed policy is superior to a Dynamic-PUT policy as long as the item characteristics vary. When the target schedule has some idle time, the Dynamic-PUT policy can outperform the optimal **u**Fixed policy by a substantial amount.



characteristics, as long as the target schedule has a small amount of idle time available.

3.2. Surplus Recovery from a Single Shock

When shocks leave the system with a surplus of inventory, we explore how the choice of the velocity parameter v affects recovery. We find that with small v, recovery is less aggressive (fewer setups and larger batches), while with larger v, recovery is more aggressive (frequent setups and smaller batches).

LEMMA 2. *If at some iteration the next item up for production is in a surplus, then a Dynamic-PUT policy will idle during this iteration an amount strictly more than the target idle time.*

Figures 4 and 5 show two different simulation runs, differing only in the value of the parameter v. Each run begins with the inventory levels of each item near B. When v is small (Figure 4), batch sizes are large and thus setups

Figure 4. $v = 2$.

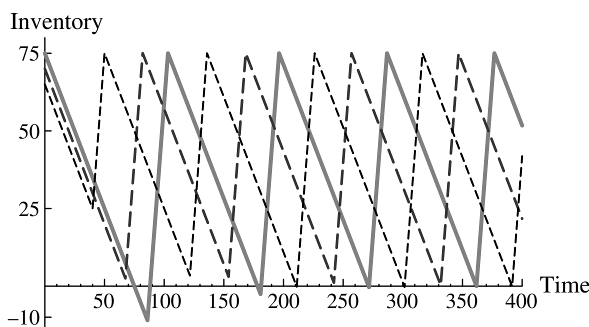
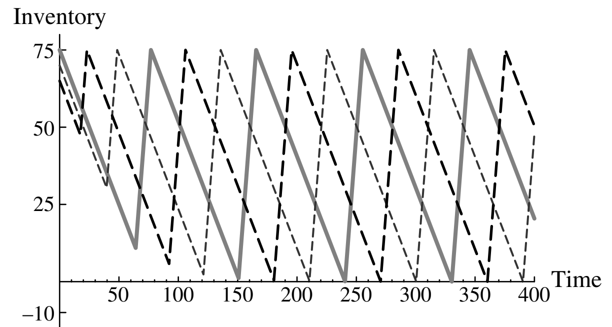


Figure 5. $v = 20$.



less frequent, but backorders are more frequent. In contrast, when v is large (Figure 5), setups are more frequent as batches are smaller, but backorders are rare.

We can bound how small production batch sizes can be as a function of v.

LEMMA 3. *The smallest production batch size possible for item i is given by*

$$B_i - \left(1 - \frac{1}{v_i}\right) \sum_{q \neq i} \tau_q + s_i.$$

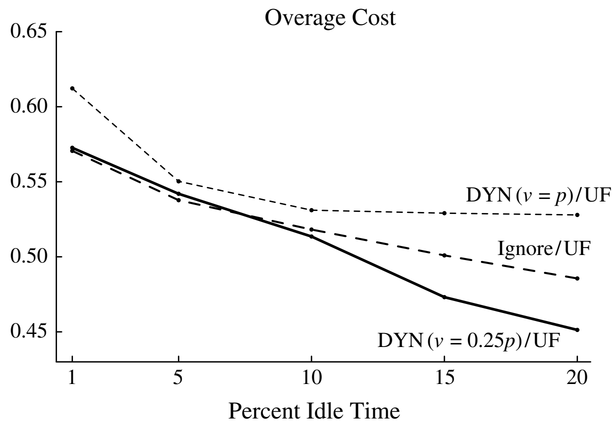
3.3. Recovery from Multiple Shocks

We now compare policies when variance is more pervasive by simulating multiple shocks over time. Here it is appropriate to consider not only costs, but how well the system maintains relative and absolute stability.

We use a discrete simulation to compare a **u**Fixed-PUT policy, a **u**Ignore-PUT, and Dynamic-PUT policies with $1 < v \leqslant p$ and identical items. We spread all available idle time evenly throughout the target schedule. Our simulation generates shocks independently with probability γ for each item at each time period of our cycle length T. When a demand shock occurs for an item, the size of the shock follows a uniform random variable in $[2, \ldots, M]$, where M is a parameter to our model and we recall that our demand for each item for each time period in the absence of a shock is normalized to 1. Thus, shocks become more frequent as γ increases and larger as M increases.

There are many parameters to such a model. We simulated a wide range of parameters and found that the relative performance of the different policies was consistent given a percent idle time in the target schedule. We present the results of a typical run with 30 items, a production rate of 38, a backorder cost of 100, and a holding cost of 1. Setup times are set implicitly to achieve the desired percent of idle time in the target schedule. We generated shocks so that the probability of a shock occurring, for any item, within the entire cycle length of T was 0.15. The shock magnitude parameter, M, was set to 15% of the peak inventory up-to point; in this case $B = 107$, so $M = 16$. We simulated 1,000 production cycles using the same sequence of random shocks when comparing the different policies.

Figure 6. When a system is subjected to multiple shocks, a Dynamic-PUT policy with various v values and a **u**Ignore-PUT policy perform similarly and substantially better than a **u**Fixed-PUT policy.



3.3.1. Cost. Figure 6 compares the excess over the target cycle cost for each policy over the simulation run as the percent idle time increases along the horizontal axis. We do not add any safety stock to any of the policies. The costs in the figure are normalized so that the **u**Fixed-PUT policy has an overage cost of 1. As one expects from our observations of single shocks, the **u**Fixed policy has the largest costs because there is some idle time available. The other policies are fairly tightly clumped between about 60% of the cost of a **u**Fixed policy when there is very little idle time, and 45% when there is substantial idle time available.

Interestingly, the **u**Ignore-PUT policy does not, in general, accumulate less cost than Dynamic-PUT policies. We conjecture that the reason is that when a **u**Ignore-PUT policy leaves the system after a shock with all idle time clumped together, a subsequent shock can be very costly. For example, suppose that all the idle time is clumped before Item 1. Then, a shock to the demand of Item 2 would not only delay Item 2, resulting in backorder costs, but then also Item 3 and so on—there is no available idle time until the end of the schedule to absorb the shock. On the other hand, when idle times are redistributed evenly throughout the schedule (as in a Dynamic-PUT policy), a shock has the opportunity to be absorbed before affecting too many items. Now, of course, clumping of idle time may be beneficial if a shock hits the right item at the right time, and a **u**Ignore-PUT policy avoids the cost of redistributing idle times back to target specification. While the tradeoffs here are certainly complex, we observe that neither a **u**Ignore-PUT policy or Dynamic-PUT policy dominates the other in terms of costs when variance is more pervasive. However, overall all runs indicate that the costs incurred for **u**Ignore-PUT and Dynamic-PUT policies are close, and considerably less costly than a **u**Fixed-PUT policy as long as there is some available idle time in the target schedule.

3.3.2. Relative Stability. The managerial importance of maintaining relative stability is to maintain the target u_i buffer before the production run of each item. Thus, we measure the percent this buffer decreases for each item, for each cycle—we do not penalize if the idle time exceeds u_i. We compute the measure

$$\left(\sum_{k=1}^{N} \frac{(u_{[n]}^{(k)} - u^{(k)})^+}{u_{[n]}^{(k)}} \times 100\right) \Big/ N \qquad (13)$$

to give the average percent underage of idle times for the simulation run.

Figure 7 compares this measure of relative stability for each policy. Here we show the perfect relative stability of the **u**Fixed policy. The Dynamic-PUT policies are clumped closely together—when there is very little idle time they lose an average of 4%–7%, and as the idle time increases they maintain excellent relative stability. On the other hand, the **u**Ignore-PUT policy performs very poorly—the figure shows this as a dotted line where values have been divided by a factor of 10. Therefore, the **u**Ignore-PUT policy sacrifices about 65% of the idle time at worst, and over 25% at best.

3.3.3. Absolute Stability. We measure how close a policy maintains the target cycle length of each item. Theorem 3 established conditions on the size of a shock so that when each v_i is set to p_i, a Dynamic-PUT policy will maintain perfect absolute stability. We compute the measure

$$\left(\sum_{k=1}^{N} \frac{|\tau_{[n]}^{(k)} - \tau^{(k)}|}{\tau_{[n]}^{(k)}} \times 100\right) \Big/ N \qquad (14)$$

Figure 7. A **u**Fixed-PUT policy maintains perfect relative stability, as shown by the horizontal line at 0%. The Dynamic-PUT policies lose some relative stability when the system has very little idle time available, while a **u**Ignore-PUT policy performs very poorly—as shown by the dotted line whose values have been divided by a factor of 10.

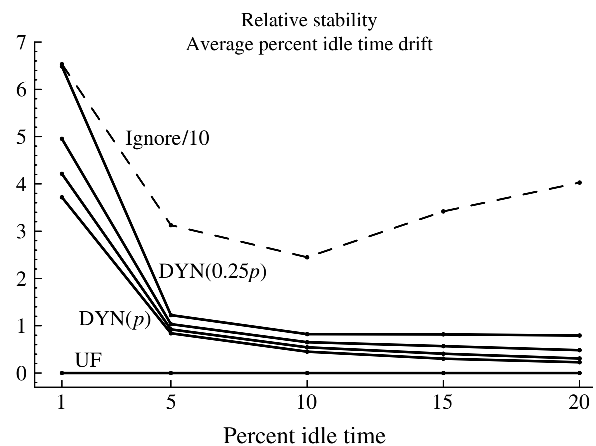
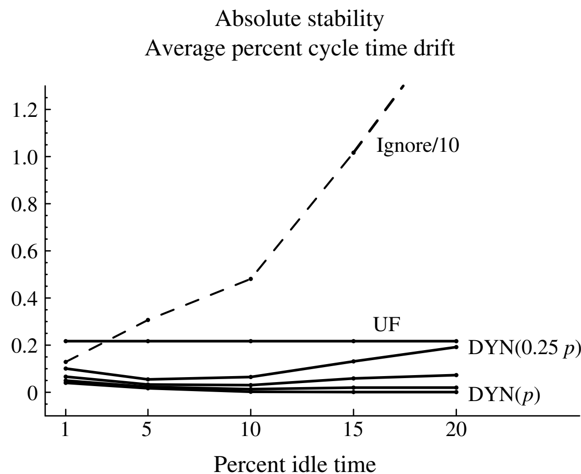


Figure 8. The Dynamic-PUT policies maintain superior absolute stability when a system is subjected to multiple shocks.



Notes. A **u**Fixed-PUT policy performs slightly worse, while a **u**Ignore-PUT policy performs very poorly—as shown by the dotted line whose values have been divided by a factor of 10.

to give the average deviation from the target item cycle lengths of $\tau_{[n]}^{(k)}$ over the length of the simulation.

Figure 8 compares the absolute stability of each policy. Here we see that the Dynamic-PUT policies are clumped tightly together, losing less than 0.2% on average. The **u**Fixed policy performs a bit worse, losing about 0.25% on average. The **u**Fixed policy maintains perfect relative stability, but is unable to use available idle time to help maintain absolute stability. On the other hand, the Dynamic-PUT policies make this trade-off, sacrificing some relative stability to help maintain better absolute stability. The **u**Ignore-PUT policy again performs poorly, as we show its results by the dotted line where values have been divided by 10. As the idle time increases, its redistribution in random ways in response to system shocks makes the cycles more and more unstable.

4. Related Work and Open Issues

Real-Time Machine Scheduling. A related stream of work considers real-time switching rules (for example, Sharifnia et al. 1990, Perkins and Kumar 1989, Chase and Ramadge 1992). These works concentrate on rules to switch a machine from production of one item to another—typically able to handle multiple machines and multiple item types with complex routings. Sharifnia et al., for example, defines corridors in inventory space that define when production switches. These switching rules, however, do not allow idling between production runs (although there may be allowance for the machine to slow; for example, Chase and Ramadge (1992) allow machines to slow to match demand). These works concentrate on whether the system will settle into a limit cycle or admit chaotic behavior—and whether the inventory level is bounded and by what degree. When applied to our single-machine cyclic scheduling setting, this stream is most closely related to **u**Fixed policies that move production from one item directly to another (because idle times are fixed). Thus, the work of Gallego (1990, 1994) defines optimal switching rules when idle times are observed. In contrast, Dynamic-PUT policies are simple with respect to their static up-to point, but allow greater flexibility in idling by adding a second trigger to determine how long to idle after switching. It is this flexible way of handling idle time that, as we saw, allows our procedure to achieve low recovery cost and good stability.

The Stochastic Economic Lot-Scheduling Problem (SELSP). Federgruen and Katalan (1996) are able to determine optimal base-stock values for the SELSP using a **u**Fixed-PUT policy. It remains an interesting open issue of how to determine the best target cyclic schedule parameters for a Dynamic-PUT policy and how its performance compares to other strategies in various stochastic settings. However, our simulations of the performance of Dynamic-PUT when there are multiple shocks indicates that it will perform very well when compared to less flexible policies.

Bourland and Yano (1994) investigate a heuristic strategy to utilize idle time in the SELSP. Their focus is on the importance of maintaining absolute schedule stability, so that each item completes production on schedule. While they allow a product to begin early by reducing its own target idle time, their system does not allow an item to use the idle time of its successor. If an item cannot be replenished within its own cycle, they assume that there is ample overtime available to recover the item. A Dynamic-PUT policy is more flexible, allowing an item to utilize available idle time of its successor and allowing items to queue for the shared resource.

Leachman and Gascon (1988) also examine a heuristic for the SELSP. Their approach is to adjust the cycle times of all the items proportionally in response to randomness to try and avoid any stockouts. Markowitz and Wein (1999) examine dynamic policies for the SELSP under heavy-traffic conditions, and provide an excellent overview of the literature.

Bucket Brigades. Our work here generalizes a model of bucket brigades (Bartholdi and Eisenstein 1996)—each item corresponds to a bucket brigade worker. Consider our system with identical items (with common target idle times) and instantaneous replenishments ($s = 0$ and $p \to \infty$). Each token value corresponds to a worker position along a bucket brigade line with each token value giving the distance of a worker to the end of the line. A token value of Z corresponds to the start of the bucket brigade line, a token value of zero to the end. During each iteration all workers (tokens) move with velocity 1 along a bucket brigade line except the last worker (token), which moves with velocity $v > 1$.

A replenishment corresponds to a bucket brigade reset: All token values remain the same except the last one, which is replenished instantaneously from zero to Z; each bucket brigade worker gets work from his predecessor except the first one, who begins a new job at the start of the line. Thus, our token system behaves exactly as a bucket brigade in which all workers have velocity 1 except the last, who has velocity $v > 1$. The self-organization of the bucket brigade corresponds to convergence of our system to the target cyclic schedule.

Thus, the Dynamic-PUT policy extends a restricted version of the normative bucket brigade model (in which all workers have the same velocity except the last, who is faster) to a cyclic scheduling system with nonidentical items and noninstantaneous replenishments. It is an open question if other more general models of a bucket brigade result in useful corresponding token algorithms for cyclic systems: For example, is it useful to consider more general sets of slowest-to-fastest worker velocities?

Selecting v_i. It is an open issue as to how best to choose the velocity parameters v_i. When the system is subjected to multiple shocks, simulations indicate that when v_i is close to each p_i, the best stability results (this follows from Theorem 3). On the other hand, smaller v may be less costly. The choice of v certainly depends on a tradeoff between stability and cost and would depend on how shocks are distributed.

5. Conclusions

Facilities that follow a cyclic schedule to replenish the inventory of a set of items by a shared resource are common. Such facilities will often insert idle time in the target schedule to hedge against uncertainties and provide periodic slack capacity for machine checks/repairs, manpower variances, etc. We introduce a new policy, a Dynamic-PUT policy, that can utilize this idle time to help maintain schedule stability and lower costs. A Dynamic-PUT policy recovers a target cyclic schedule—including the layout or distribution of idle times within the schedule—after a single disruption. However, we find that such a policy is also useful when disruptions are more pervasive.

A Dynamic-PUT policy distinguishes itself in how it is able to flexibly utilize available idle time in a target schedule during recovery; reducing or eliminating idle time whenever the system is in a deficit, or inserting idle time if the system is in a surplus. This is in contrast to two other policies. A **u**Fixed policy will recover the target schedule, but constrains itself to observe the idle times during recovery. A **u**Ignore policy ignores idle times during recovery and thus does not recover the target layout of idle times.

Our policy is shown to be very effective in comparison to optimal **u**Fixed policies. When a proportionality condition holds we prove that a Dynamic-PUT policy will recover from a deficit faster, and with less cost than the optimal **u**Fixed policy if there is any idle time available in the target schedule. A Dynamic-PUT policy performs very well against the optimal **u**Fixed recovery policy even when the proportionality condition does not hold as long as there is a small amount of idle time in the target schedule (above 1%).

The costs of a Dynamic-PUT policy compare closely with a **u**Ignore policy. Interestingly, even though a **u**Ignore policy is less constrained, we find that a Dynamic-PUT policy can display cost advantages when a system is subjected to frequent shocks. Also as one would expect, a Dynamic-PUT policy performs much better than a **u**Ignore policy in terms of maintaining schedule stability.

So overall, a Dynamic-PUT policy performs very well in terms of both cost and stability. Furthermore, a Dynamic-PUT policy allows a manager to control recovery behavior with a single "knob"—making recovery quick, aggressive, and stable; or slower and more methodical. Finally, implementation is simple, augmenting traditional produce-up-to policies with a set of counters or tokens.

Appendix A

A.1. Proof of Theorem 2: Balance

Proof. We can write from Equations (10) and (11),

$$y_{[j]}^{(k+1)} = y_{[j-1]}^{(k)} - \frac{Z_{[n]}^{(k)} + p_{[n]}^{(k)} s_{[n]}^{(k)}}{p_{[n]}^{(k)} - 1} + \frac{y_{[n]}^{(k)} - (y_{[n]}^{(k)})^{+}(p_{[n]}^{(k)}/v_{[n]}^{(k)})}{p_{[n]}^{(k)} - 1} \quad \text{for } j = 2, \ldots, n. \tag{15}$$

Then, substituting the equation for $y_{[1]}^{(k-(n-2))}$ into $y_{[2]}^{(k-(n-3))}$, then into $y_{[3]}^{(k-(n-4))}$, and so on, we derive

$$y_{[n]}^{(k+1)} = Z_{[n]}^{(k+1)} - \sum_{q=k-(n-2)}^{k} \frac{Z_{[n]}^{(q)} + p_{[n]}^{(q)} s_{[n]}^{(q)}}{p_{[n]}^{(q)} - 1} + \sum_{q=k-(n-2)}^{k} y_{[n]}^{(q)} a^{(q)}, \tag{16}$$

where

$$a^{(q)} = \begin{cases} \dfrac{1}{p_{[n]}^{(q)} - 1} & \text{if } y_{[n]}^{(q)} \leqslant 0, \\ \dfrac{1 - (p_{[n]}^{(q)}/v_{[n]}^{(q)})}{p_{[n]}^{(q)} - 1} & \text{if } y_{[n]}^{(q)} > 0. \end{cases}$$

If we let $\boldsymbol{y}^{(k)} = (y_{[n]}^{(k-(n-2))}, \ldots, y_{[n]}^{(k-1)}, y_{[n]}^{(k)})$, we can write the dynamics of the system as

$$\boldsymbol{y}^{(k+1)} = A^{(k)} \boldsymbol{y}^{(k)} + \mathbf{r}^{(k)}, \tag{17}$$

where we define the $n-1$-dimensional vector

$$\mathbf{r}^{(k)} = \left(0, 0, \ldots, 0, Z_{[n]}^{(k+1)} - \sum_{q=k-(n-2)}^{k} \frac{Z_{[n]}^{(q)} + p_{[n]}^{(q)} s_{[n]}^{(q)}}{p_{[n]}^{(q)} - 1}\right)$$

and the $(n-1)\times(n-1)$ dimensional matrix

$$A^{(k)}=\begin{pmatrix} 0 & 1 & 0 & \cdots & & & 0\\ 0 & 0 & 1 & 0 & \cdots & & 0\\ 0 & 0 & 0 & 1 & 0 & \cdots & 0\\ & & \ddots & & & \ddots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1\\ a^{(k-(n-2))} & & & & \cdots & a^{(k-1)} & a^{(k)} \end{pmatrix}.$$

We show that $A^{(\infty)}\cdots A^{(k+1)}A^{(k)}\to 0$. We first note that, because each $v_{[n]}^{(q)}\geqslant p_{[n]}^{(q)}$, then each $0\leqslant a^{(q)}\leqslant 1/(p_{[n]}^{(q)}-1)$. Thus, each $A^{(k)}$ is nonnegative, and thus to show convergence it suffices to consider the case where the matrix entries are their largest. This occurs when each $y_{[n]}^{(q)}\leqslant 0$, and thus each $a^{(q)}=1/(p_{[n]}^{(q)}-1)$.

That is, we show that the matrices are convergent whenever all token values are negative—the system is in a severe deficit in that there is a queue of items waiting for production at the shared resource. The convergence then follows from our Slack Capacity Assumption 1. Queueing must dissipate because each batch size produced is above the target level, and thus, in comparison to the target schedule, more time is spent producing. This is shown formally in Gallego (1994) for convergence of the **u**Fixed-PUT system, so we do not repeat the details here. However, the argument is to consider an extra unit of production (over the target lot size) of item i, and consider that the system gains $1/p_i$ in production time, but sacrifices during this time, due to demand, $(1/p_i)(\sum_q 1/p_q)$ production time over all the items. However, by our Slack Capacity Assumption 1, $\sum_q 1/p_q<1$, and thus the system will gain more than it will lose. Thus, any production cycle through the n items in which the batch sizes exceed the target will gain on the system. Furthermore, with Equation (12), as in Gallego (1994), queueing ($y_{[n]}^{(q)}\leqslant 0$) is transient. Therefore, $A^{(\infty)}\cdots A^{(k+1)}A^{(k)}\to 0$, and in particular, every n length subsequence $A^{(k+(n-1))}\cdots A^{(k+1)}A^{(k)}$ has all eigenvalues within the unit disk, and the $\mathbf{r}^{(k)}$ cycle every n iterations. □

A.2. Proof of Theorem 3

Proof.

$$\begin{aligned} x_{[n]}^{(k)} &> \hat{x}_{[n]}^{(k)}-u_{[n]}^{(k)}v_{[n]}^{(k)}\\ &= u_{[n]}^{(k)}-u_{[n]}^{(k)}v_{[n]}^{(k)} \quad \text{by definition of Equations (10)}\\ &= u_{[n]}^{(k)}(1-v_{[n]}^{(k)})=B_{[n]}^{(k)}-Z_{[n]}^{(k)} \quad \text{from Equation (8);} \end{aligned}$$

and therefore $y_{[n]}^{(k)}>0$ from Equation (11). We can now rewrite Equation (9) using Equations (8) and (11) to get

$$\tau^{(k)}=y_{[n]}^{(k)}/v_{[n]}^{(k)} + \frac{B_{[n]}^{(k)}-y_{[n]}^{(k)}+u_{[n]}^{(k)}(v_{[n]}^{(k)}-1)+y_{[n]}^{(k)}/v_{[n]}^{(k)}+p_{[n]}^{(k)}s_{[n]}^{(k)}}{p_{[n]}^{(k)}-1};$$

which when $v_{[n]}^{(k)}=p_{[n]}^{(k)}$, yields

$$\tau^{(k)}=u_{[n]}^{(k)}+\frac{B_{[n]}^{(k)}+p_{[n]}^{(k)}s_{[n]}^{(k)}}{p_{[n]}^{(k)}-1}=\tau_{[n]}^{(k)}.$$

(In short, a δ shock means δ/v less (more) idle time and $(\delta/v)/(p-1)$ less (more) production time due to accumulated demand during idling, but $\delta/(p-1)$ more (less) production time to recover the shock—these terms cancel out when $v=p$, resulting in the preservation of the target cycle time.) Because the duration of the iteration was precisely on target, $x_{[n]}^{(k+1)}>\hat{x}_{[n]}^{(k+1)}-u_{[n]}^{(k+1)}v_{[n]}^{(k+1)}$, and therefore $\tau^{(k+1)}=\tau_{[n]}^{(k+1)}$ and so on. Thus, after n iterations (a production of each item) the target is reached. □

A.3. Proof of Theorem 4: Imbalance

Proof. When each $v_i=1$, the vectors $\mathbf{x}^{(k)}$ and $\mathbf{y}^{(k)}$ are identical at each iteration k, and so we can present our proof using only the $\mathbf{x}^{(k)}$ vector.

From System (10), we have

$$\tau^{(k)}=\left\{\begin{array}{ll} \dfrac{B_{[n]}^{(k)}-x_{[n]}^{(k)}+p_{[n]}^{(k)}s_{[n]}^{(k)}}{p_{[n]}^{(k)}-1} & \text{if } x_{[n]}^{(k)}\leqslant 0,\\ x_{[n]}^{(k)}+\dfrac{B_{[n]}^{(k)}+p_{[n]}^{(k)}s_{[n]}^{(k)}}{p_{[n]}^{(k)}-1} & \text{if } x_{[n]}^{(k)}>0. \end{array}\right\} \tag{18}$$

The cycle time of an iteration can also be expressed as

$$\tau^{(k)}=x_{[j-1]}^{(k)}-x_{[j]}^{(k+1)} \quad \text{for any } j=2,\ldots,n,$$

and therefore,

$$x_{[1]}^{(k)}=B_{[n]}^{(k-1)},$$

$$x_{[j]}^{(k)}=B_{[n]}^{(k-j)}-\sum_{q=k-(j-1)}^{k-1}\tau^{(q)} \quad \text{for } j=2,\ldots,n. \tag{19}$$

Suppose that the system exhibits a d-cycle in cycle time space; then, from Equations (19) it is clear that the $x_{[2]}^{(k)}$ component will also exhibit a d-cycle, and thus the cyclic behavior we observe in cycle time space will manifest itself directly in inventory space.

Substituting (19) into (18), we have

$$\tau^{(k)}=\left\{\begin{array}{l} \dfrac{1}{p_{[n]}^{(k)}-1}(\tau^{(k-(n-1))}+\cdots+\tau^{(k-2)}+\tau^{(k-1)})+\dfrac{p_{[n]}^{(k)}s_{[n]}^{(k)}}{p_{[n]}^{(k)}-1}\\ \qquad \text{if } x_{[n]}^{(k)}\leqslant 0,\\ -\tau^{(k-(n-1))}-\cdots-\tau^{(k-2)}-\tau^{(k-1)}+\dfrac{p_{[n]}^{(k)}(B_{[n]}^{(k)}+s_{[n]}^{(k)})}{p_{[n]}^{(k)}-1}\\ \qquad \text{if } x_{[n]}^{(k)}>0. \end{array}\right.$$

Letting $\boldsymbol{\tau}^{(k)}$ be the $n-1$-dimensional vector $(\tau^{(k-(n-2))}, \ldots, \tau^{(k-1)}, \tau^{(k)})$, we can write the dynamics of our linear system as

$$\boldsymbol{\tau}^{(k+1)} = A^{(k)}\boldsymbol{\tau}^{(k)} + \mathbf{r}^{(k)},$$

where we define the $n-1$-dimensional vector

$$\mathbf{r}^{(k)} = \begin{cases} \left(0, 0, \ldots, 0, \dfrac{p_{[n]}^{(k)} s_{[n]}^{(k)}}{p_{[n]}^{(k)} - 1}\right) & \text{if } x_{[n]}^{(k)} \leqslant 0, \\ \left(0, 0, \ldots, 0, \dfrac{p_{[n]}^{(k)} (B_{[n]}^{(k)} + s_{[n]}^{(k)})}{p_{[n]}^{(k)} - 1}\right) & \text{if } x_{[n]}^{(k)} > 0, \end{cases}$$

and if $x_{[n]}^{(k)} \leqslant 0$, we have the $(n-1) \times (n-1)$-dimensional matrix

$$A^{(k)} = \begin{pmatrix} 0 & 1 & 0 & \cdots & & & 0 \\ 0 & 0 & 1 & 0 & \cdots & & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 \\ & & & \ddots & & \ddots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 \\ \frac{1}{p_{[n]}^{(k)}-1} & \frac{1}{p_{[n]}^{(k)}-1} & & \cdots & & \frac{1}{p_{[n]}^{(k)}-1} & \frac{1}{p_{[n]}^{(k)}-1} \end{pmatrix},$$

and if $x_{[n]}^{(k)} > 0$, we have the $(n-1) \times (n-1)$-dimensional matrix

$$A^{(k)} = \begin{pmatrix} 0 & 1 & 0 & \cdots & & & 0 \\ 0 & 0 & 1 & 0 & \cdots & & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 \\ & \ddots & & & & \ddots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 \\ -1 & -1 & & \cdots & & -1 & -1 \end{pmatrix}.$$

If there is a queue for production ($x_{[n]}^{(k)} \leqslant 0$), then the sequence $A^{(k)} \cdots A^{(1)} A^{(0)} \to 0$ as k gets large. This follows in the same manner as the convergence of Theorem 2 detailed in Appendix A.1—because our system has some slack capacity, any overall deficit will dissipate.

Now, furthermore, once the $x_{[n]}^{(k)}$s are positive they will remain there. This follows from Equation (18), because once queueing ceases, production runs have separated the inventory levels of successive items (the cycle time) by at least $(B_{[n]}^{(k)} + p_{[n]}^{(k)} s_{[n]}^{(k)})/(p_{[n]}^{(k)} - 1)$, and with each $v_i = 1$ the separation remains static. Therefore, each $x_{[n]}^{(k)}$ will remain positive.

With all $x_{[n]}^{(k)}$ positive, we now just write A and $\mathbf{r}^{(k)}$ to describe the dynamics, and we reset our iteration counter to 0. We have

$$\boldsymbol{\tau}^{(k+1)} = A^{k+1}\boldsymbol{\tau}^{(0)} + (I + A + A^2 + \cdots + A^k)\mathbf{r}^{(k)}. \tag{20}$$

First, we have $A^n = I$. Therefore, the sequence $\{A, A^2, \ldots, A^k, \ldots\}$ cycles every n iterations, and thus the term $A^{k+1}\boldsymbol{\tau}^{(0)}$ from Equation (20) will cycle every n iterations. Furthermore, each matrix in the sequence $\{A, A^2, \ldots, A^k, \ldots\}$ is composed of $n-1$ row vectors (each of dimension $n-1$) that cycle through the n vectors $(-1, -1, \ldots, -1)$, $\mathbf{e}^1$, $\mathbf{e}^2, \ldots, \mathbf{e}^{n-1}$, where $\mathbf{e}^q$ is the unit vector of all zeroes except a 1 in the qth position. A^q is composed of all the row vectors except $\mathbf{e}^q$ for $q = 1, \ldots, n-1$; and A^n is composed of all the row vectors except $(-1, -1, \ldots, -1)$. Then, the cycle repeats. From this structure, we note that the sum of any n consecutive terms of $\{A, A^2, \ldots, A^k, \ldots\}$ sum to the zero matrix,

$$\sum_{i=1}^{n} A^{q+i} = 0 \quad \text{for any } q = 0, 1, \ldots.$$

Thus, the last term of Equation (20) also cycles every n iterations. Thus, the $\tau^{(k)}$ cycle through one of an infinite number of possible n-cycles, and from Equations (19) the inventory levels also cycle. This system may exhibit some d-cycle for some $d < n$, if by chance some entries of $\boldsymbol{\tau}^{(k)}$ are the same. However, when $x_{[n]}^{(k)} > 0$, the eigenvalues of A are all 1, so the cycles are not stable—any perturbation will throw the system into a new d-cycle, $d \leqslant n$. □

Acknowledgments

The author thanks the anonymous reviewers for their helpful suggestions and insights—their suggestions have greatly improved the exposition. In addition, he thanks Dan Adelman, John Bartholdi, Gary Eppen, and Anuj Mehrotra for helpful discussions and comments on earlier drafts.

The author also thanks the management at Allied Foods, Inc. in Atlanta, Georgia for sharing their cyclic production system and insights. Furthermore, he acknowledges the support of the Graduate School of Business at The University of Chicago.

Portions of this work have been presented at The University of Chicago, Cornell University, the University of Pennsylvania, Purdue University, and the University of Toronto. The author thanks the audiences for many interesting and stimulating questions.

References

Bartholdi, J. J., III, D. D. Eisenstein. 1996. A production line that balances itself. *Oper. Res.* **44** 21–34.

Bourland, K., C. Yano. 1994. The strategic use of capacity slack in the economic lot scheduling problem with random demand. *Management Sci.* **40** 1690–1704.

Chase, C., P. J. Ramadge. 1992. On real-time scheduling policies for flexible manufacturing systems. *IEEE Trans. Automat. Control* **37** 491–496.

Delporte, C., L. Thomas. 1997. Lot sizing and sequencing for N products on one facility. *Management Sci.* **23** 1070–1079.

Dobson. G. 1987. The economic lot-scheduling problem: Achieving feasibility using time-varying lot sizes. *Oper. Res.* **35** 764–771.

Federgruen, A., Z. Katalan. 1996. The stochastic economic lot scheduling problem: Cyclical base-stock policies with idle times. *Management Sci.* **42** 783–796.

Gallego, G. 1990. Scheduling the production of several items with random demands in a single facility. *Management Sci.* **36** 1579–1592.

Gallego, G. 1994. When is a base stock policy optimal in recovering disrupted cyclic schedules? *Naval Res. Logist.* **41** 317–333.

Leachman, R. C., A. Gascon. 1988. A heuristic scheduling policy for multi-item, single machine production systems with time-varying stochastic demands. *Management Sci.* **34** 377–390.

Markowitz, D. M., L. M. Wein. 2001. Heavy traffic analysis of dynamic cyclic policies: A unified treatment of the single machine scheduling problem. *Oper. Res.* **49** 246–270.

Maxwell. W. L. 1964. The scheduling of economic lot sizes. *Naval Res. Logist. Quart.* **11** 89–124.

Perkins, J. R., P. R. Kumar. 1989. Stable, distributed, real-time, scheduling of flexible manufacturing/assembly/disassembly systems. *IEEE Trans. Automat. Control* **34** 139–148.

Schonberger, R. J. 1982. *Japanese Manufacturing Techniques: Nine Hidden Lessons in Simplicity*. The Free Press, New York, 68–69.

Sharifnia, A., M. Caramanis, S. B. Gershwin. 1990. Dynamic setup scheduling and flow control in manufacturing systems. *Discrete Event Dynamic Systems: Theory Appl.* **34** 139–148.

Sox, C. R., P. L. Jackson, A. Bowman, J. A. Muckstadt. 1999. A review of the stochastic lot scheduling problem. *Internat. J. Production Econom.* **62** 181–200.

Suzaki, K. 1987. *The New Manufacturing Challenge: Techniques for Continuous Improvement*. The Free Press, New York, 113–123.

Zipkin, P. 1991. Computing optimal lot sizes in the economic lot scheduling problem. *Oper. Res.* **39**(1) 56–63.