Comput. & Indus. Engng Vol. 9, No. 4, pp. 387–393, 1985
Printed in the U.S.A.

0360-8352/85 $3.00 + .00

COMPUTING TRANSFER LINES PERFORMANCE MEASURES USING DYNAMIC PROGRAMMING

PIERRE L'ECUYER
Université Laval, Ste-Foy, Québec, Canada G1K 7P4

(*Received for publication* 12 *March* 1985)

Abstract—The dynamic behavior of a three-stage production transfer line with finite buffers and unreliable machines is modeled as a Markov chain. Performance measures are computed using value iteration dynamic programming with over-relaxation steps and finite element approximation. This approach permits considering larger buffers than is possible when solving directly for the steady-state probabilities.

1. INTRODUCTION

A transfer line is a sequence of machines (or stages) connected by transfer mechanisms (see Fig. 1). Items enter from one end of the line, visit each machine, and emerge at the other end. In the model discussed here, the machines are liable to failure at random times, and remain inoperative for random durations while they are under repair. To decrease the effects of machine failures on the rest of the line, buffer storages are placed between the unreliable machines. Larger buffers improve the line throughput, but their costs in terms of larger inventories, floor space and additional equipment must be taken into account. Applications of transfer lines models arise in a wide range of areas, including manufacturing industry, computer science, mining, etc. (see [4, 6, 8]).

Transfer lines have been studied by many authors during the last 30 years. See [6] for a list of references. For given storage allocations, these authors wish to compute various performance measures, such as the line throughput or the mean total inventory. Some also seek to distribute a fixed total buffer size to the $k - 1$ buffers so as to maximize the line throughput.

For two-stage lines, the throughput can be computed analytically[3, 6]. Gershwin and Schick[6] proposed a Markov chain model for a 2- or 3-stage line, and a method for solving the large linear system for the steady-state probabilities. The method is easy to implement for the two-machines case, but for the three-machines case, computer time and memory requirements increase respectively as the cube and square of the total buffer capacity. Also, even with 35 decimal digits extended precision computation, numerical instabilities prevent any reliable answers when the buffer capacities are larger than about 15. Other analytic techniques for more than two unreliable machines and finite buffers have been proposed only for models with more restricted assumptions than those retained in [6].

Lines with more than three stages have been studied by approximation[3, 5, 9], or by simulation[8,13].

In this paper, we use the same Markov chain model as in [6]. However, instead of computing the steady-state probabilities, we compute the throughput and mean total inventory via value iteration dynamic programming. We also experiment with different methods to approximate the value function, and over-relaxation steps to accelerate the convergence. This approach permits us to cope with larger buffers than in [6].

2. THE MODEL

Many variants of the transfer line model appear in the literature. Here, we adopt the same variant as in [6].

The system is comprised of k machines, serially ordered and separated by $k - 1$ finite capacity buffers (see Fig. 1). An inexhaustible supply of items is available to

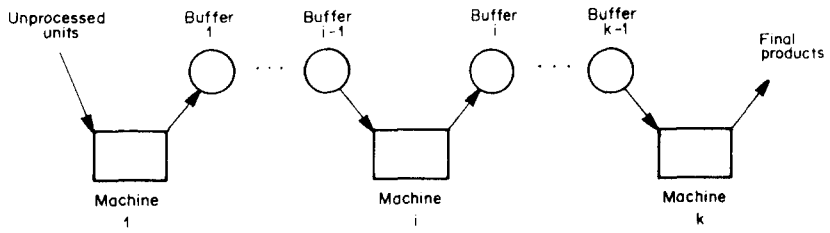

Fig. 1. Diagram of a k-stage transfer line.

machine 1, and items are brought into an unlimited storage area after being processed by machine k. All machines are synchronized, and have equal and constant service times. That common one time unit fixed cycle includes transportation time to or from the buffers.

Items move from machine 1 to buffer 1, . . . , to buffer $i-1$, to machine i, to buffer i, . . . , to machine k. Each machine in the line may fail at a random time and for a random duration. Actually, a 'failure' may correspond to stoppage for maintenance, adjustments, repairs, etc. . . . Machines are assumed to have geometrically distributed number of processing cycles between failures and number of cycles to repair. If machine i is processing an item during a cycle, there is a probability p_i that it fails during that cycle. If machine i is down (in failure state) at the beginning of a cycle, there is a probability r_i that it is repaired during that cycle.

For $i = 1, \ldots, k-1$, let N_i be the capacity of buffer i. When machine i is down, the level in buffer $i-1$ may rise. If the failure persists, that buffer may fill up and force machine $i-1$ to stop its processing. In such a case, machine $i-1$ will be called blocked. Similarly, the level in buffer i may fall, as machine $i+1$ drains its contents. That buffer may empty and force machine $i+1$ to idleness. Such a forced down machine will be called *starved*. If a failure persists long enough, the blocking and starvation effects could propagate up and down the line. Notice that machine 1 can never be starved and machine k can never be blocked. The role of the buffers is to provide a partial decoupling of the machines and lessen the effects of a failure.

It is assumed that machines can fail only while processing items. A starved or blocked machine cannot fail. When a machine fails, the item it was working on is not damaged or destroyed, but is returned to the previous storage location. Processing on that item is resumed when the machine is repaired.

We assume that a cycle begins with the transitions in the machine conditions (breakdown or repair) and ends with storage levels modifications that depend on the new machine states. Notice that with this convention, a machine that is repaired during a cycle is assumed to be operational for that cycle and can process an item (i.e. finish the processing cycle that was aborted previously). Actually, processing can take place on machine i only if at the beginning of the cycle, there is at least one item in buffer $i-1$ and at least one free location in buffer i. If this is the case and machine i does not fail, one item is removed from buffer $i-1$ and added to buffer i at the end of the cycle.

During a cycle, we say that the transfer line is *up* if machine k successfully processes an item. Otherwise, it is down. The throughput or efficiency E of the line is the steady-state proportion of up cycles, that is the mean number of items coming out of the line per cycle, in the long run. The mean in-process inventory I is the expected total number of units in the buffers at any given cycle, in steady-state. We wish to compute E and I, for given values of $k, N_1, \ldots, N_{k-1}, p_1, \ldots, p_k, r_1, \ldots, r_k$.

For $i = 1, \ldots, k$, the condition of machine i is given by

$$a_i = \begin{cases} 0 & \text{if machine } i \text{ is down} \\ 1 & \text{if machine } i \text{ is operational.} \end{cases} \tag{1}$$

Notice that an operational machine can be either processing, blocked or starved. For $i = 1, \ldots, k-1$, let n_i be the number of items in buffer i. The state of the system

is $s = (n_1, \ldots, n_{k-1}, a_1, \ldots, a_k)$, where $a_i = 0$ or 1 and $0 \leqq n_i \leqq N_i$ for each i. The cardinality of the state space S is

$$M = 2^k \prod_{i=1}^{k-1} (N_i + 1). \tag{2}$$

3. A VALUE ITERATION APPROACH

The model can be seen as a discrete-time Markov chain with state dependent 'returns', where E and I are average values per cycle.

We define the one cycle value functions g: $S \to [0, 1]$ and f: $S \to \{0, 1, 2, \ldots\}$ as

$$g(s) = \begin{cases} 0 & \text{if } n_{k-1} = 0 \\ r_k & \text{if } n_{k-1} > 0 \text{ and } a_k = 0 \\ 1 - p_k & \text{if } n_{k-1} > 0 \text{ and } a_k = 1 \end{cases} \tag{3}$$

$$f(s) = n_1 + n_2 + \cdots + n_{k-1}. \tag{4}$$

The system being in state s at the beginning of a cycle, $g(s)$ is the probability that an item emerges from the line at the end of that cycle, and $f(s)$ is the in-process inventory during that cycle.

For s and x in S, let $p(s, x)$ be the transition probability from state s to state x in one cycle. The computation of these probabilities is cumbersome, but straightforward. They are analyzed more fully in [6]. Notice that the underlying Markov chain has a closed irreducible set of states, plus some transient states. For instance, $s = (0, 0, \ldots, 0)$ is transient, since buffer $i - 1$ cannot be empty if machine i is failed.

In what follows, we discuss only the computation of E. The same type of reasoning is also valid for I.

Theoretically, E can be computed by the following algorithm, due to White[12]:

begin
pick a recurrent state y in S;
set $h(s) := 0$ for all s in S;
repeat
$H(s) := g(s) + \sum_{x \in S} p(s, x)h(x)$, all s in S; (5)
$d^+ := \max_{s \in S} (H(s) - h(s))$; (6)
$d^- := \min_{s \in S} (H(s) - h(s))$; (7)
$h(s) := H(s) - H(y)$, all s in S (8)
until
$d^+ - d^-$ is small enough
end.

Theorem 1. After each iteration of the repeat loop, we have $d^- \leqq E \leqq d^+$. Furthermore, the successive values of d^- and d^+ converge to E. Hence, if 'small enough' means smaller than ϵ, for $\epsilon > 0$, then the algorithm converges in a finite number of iterations.

Proof: It follows from propositions 6 and 7 in chapter 8 of [2].

Unfortunately, when S is very large, this algorithm could be very time consuming, or even impossible to implement on a computer in its present form. One may look for approximations. For instance, instead of computing $H(s)$ for every state, compute it only for a small subset of S, and interpolate for the remaining states. This technique has been studied in [7] for the case of discounted d.p.

Another problem with the algorithm is that its convergence is very slow, especially

when the buffer sizes are large. For a given s, the sequence of values of $H(s)$ behave somewhat like a geometric sequence, but with a ratio of almost one. Various modifications can be tried to accelerate the convergence. One of them is to perform *over-relaxation* steps. Such a step consists of replacing equation (8) by

$$h(s) := h(s) + \tau\,[H(s) - H(y) - h(s)], \text{ all } s \text{ in } S \tag{9}$$

where $\tau \geqq 1$ is the relaxation factor. The step can be performed every say λ iterations. One can also vary the values of λ and τ during the execution.

The idea of over-relaxation was considered in [10] and [11] for the case of discounted dynamic programming. Equation (9) was used at every iteration ($\lambda = 1$), and τ was restricted to be in the interval (0, 2). Obviously, without restrictions on τ and λ, the algorithm does not necessarily converge. But for a fixed λ, if one uses a decreasing sequence of values of τ converging to a value smaller than one, the convergence is certainly guaranteed.

More interestingly, one could implement the algorithm through an interactive procedure, where the user can observe the successive values of d^+ and d^-, halt the execution after any iteration to contemplate a graphical display of H or h, adjust the values of λ and τ accordingly or change the method of approximation, and even return to a previous iteration (i.e. previous value of h). This is the approach we took, and it is illustrated in the next section for three-machine examples. The programs were implemented in FORTRAN on a VAX-11/780 mini computer.

4. NUMERICAL ILLUSTRATIONS

Consider a three-stage transfer line ($k = 3$). The state of the system is denoted $s = (n_1, n_2, a_1, a_2, a_3)$. In equation (5), one may compute $H(s)$ for all 8 possible values of $a = (a_1, a_2, a_3)$, but only on a grid of values of (n_1, n_2). More precisely, choose two positive integers α and β, and integers u_i and v_j such that

$$\begin{aligned} 0 = u_1 < u_2 < \cdots < u_\alpha &= N_1 \\ 0 = v_1 < v_2 < \cdots < v_\beta &= N_2. \end{aligned} \tag{10}$$

Let

$$S_* = \{(u_i, v_j, a),\ 1 \leqq i \leqq \alpha,\ 1 \leqq j \leqq \beta,\ a \in \{0, 1\}^3\}.$$

At each iteration, compute $H(s)$ on S_*, and approximate for the other states, using an approximation surface for each value of a. Four approximation methods were implemented in the programs.

Method 1 is piece-wise bilinear interpolation (BILIN). For each a, the points of S_* determine a partition of S into $(\alpha - 1)(\beta - 1)$ subrectangles, and we compute an interpolating function which is bilinear on each subrectangle, i.e. has the form

$$F(n_1, n_2, a) = c_{ija} + c(1)_{ija} n_1 + c(2)_{ija} n_2 + c(1, 2)_{ija} n_1 n_2,$$
$$u_i \leqq n_1 \leqq u_{i+1} \text{ and } v_j \leqq n_2 \leqq v_{j+1}. \tag{11}$$

Method 2 is piece-wise biquadratic interpolation (BIQUAD). It takes the subrectangles by blocks of four, and the interpolating function is quadratic in n_1 and n_2, on each block. α and β must be odd, and the interpolating function has the form

$$\begin{aligned} F(n_1, n_2, a) = c &+ c(1)n_1 + c(1, 1)n_1^2 + c(1, 2)n_1 n_2 \\ &+ c(2)n_2 + c(2, 2)n_2^2 + c(1, 1, 2)n_1^2 n_2 \\ &+ c(1, 2, 2)n_1 n_2^2 + c(1, 1, 2, 2)n_1^2 n_2^2 \end{aligned} \tag{12}$$

on each of the $(\alpha - 1)(\beta - 1)/4$ pieces.

Method 3 is piece-wise bicubic interpolation (BICUB), as proposed by Akima[1]. It asks for 16 coefficients on each of the $(\alpha - 1)(\beta - 1)$ pieces.

The last method uses piece-wise constant approximation (PCONST), which is equivalent to approximation by state aggregation, as described in [2].

Notice that the size of the grid, or the method of approximation, may vary from iteration to iteration. It should be reasonable to start with a coarse grid and the simplest method, refining the mesh gradually and switching to more sophisticated approximation methods as the iterative process goes on. All these methods converge in the following sense: when α attains N_1 and β attains N_2, we have $S_* = S$ and the error of approximation becomes 0. One will note that d^- and d^+ provide bounds for the limiting value of $H(y)$, using the current approximation scheme. To obtain real bounds on E, one must take $S_* = S$ for one iteration at the end.

The algorithm now operates as follows:

Algorithm
begin
read N_1, N_2, p_i and r_i for $i = 1, 2, 3$, and y;
set $h(s)$ to 0 for all s in S;
repeat
read α, β, $u_1, \ldots, u_\alpha$, $v_1, \ldots, v_\beta$, τ, λ, an approximation method, and a positive integer NITER;
for $n := 1$ *to* NITER *do begin*
compute (5–7) for S_* instead of S;
if λ divides NITER *then*
compute (9) for S_*
else
compute (8) for S_*;
print $H(y)$, d^- and d^+;
end
until no more data to read;
end.

Example 1. Let $N_1 = N_2 = 10$, $p_i = 0.01$ and $r_i = 0.09$ for $i = 1, 2, 3$, and $y = (5, 5, 1, 1, 1)$. Taking $\alpha = \beta = 5$, $(u_1, \ldots, u_5) = (v_1, \ldots, v_5) = (0, 2, 4, 7, 10)$, performing 400 iterations with $\tau = 1$ we obtain the results of Table 1. For PCONST[1], we took the average of the four corners as the approximation value on each subrectangle, while for PCONST[2], we took the lower left corner. We see that PCONST yields very bad results, while for the other three methods, $d^+ - d^-$ converge to 0 and the results are very consistent. If we take the results of either BILIN, BIQUAD or BICUB (the resulting h) and perform another 400 iterations with the whole state space ($\alpha = \beta = 11$), we obtain $H(y) = 0.79977 = d^- = d^+$, which is the true value of E. The programs were run with simple, double, and extended (35 decimal digits) precision arithmetic, and this same value was obtained in all cases. For the same problem, Gershwin and Schick[6] obtained $E = 0.8000$ (which is exact to 3 decimal places) using extended precision arithmetic.

Notice that the approximation obtained with BIQUAD or BICUB is quite good. BILIN yields an underestimation of E, and it is due to the fact that h is concave.

Experiments with over-relaxation were also done with this example, and typically good values for λ and τ were $\lambda \cong 10$ and τ between 10 and 20. With such values, it

Table 1. Results for example 1 with a 5 by 5 approximation grid

Approx. method	d^-	$H(y)$	d^+
BILIN	.7851	.7852	.7852
BIQUAD	.7945	.7946	.7946
BICUB	.7951	.7952	.7952
PCONST^1	-.046	.883	1.185
PCONST^2	.000	.001	5.025

took about 2 to 3 times fewer iterations to get the same value for $d^+ - d^-$, compared to what it took using $\tau = 1$.

Table 2 provides a typical illustration of what goes on. Note that the iteration numbers for which the results are shown have been carefully chosen not to coincide with over-relaxation steps. Each of the these steps induce a perturbation of the function $h(\cdot)$, which sometime deteriorates the tightness of the bounds d^- and d^+ at the corresponding iteration. A typical pattern appears in Table 3.

Example 2. Let $N_1 = 40$, $N_2 = 60$, $p_1 = 0.03$, $p_2 = 0.05$, $p_3 = 0.08$, $r_1 = r_2 = r_3 = 0.20$, and $y = (20, 30, 1, 1, 1)$. For this case, due to numerical instabilities, it would be practically impossible to solve the linear system for the steady state probabilities. For instance, this example cannot be solved using the Gershwin and Schick's method. However, E can be computed using d.p. and successive grid refinements.

Table 2. Intermediate results for example 1 with a 5 × 5 grid. The values shown are those of d^-, $H(y)$ and d^+, in that order. Note that the convergence is not monotonous

iteration	BILIN τ=1			BILIN τ=10, λ=10		
49	.75619	.78422	.81676	.77656	.78646	.79787
99	.77482	.78413	.79748	.78394	.78511	.78700
149	.78115	.78465	.78998	.78499	.78516	.78545
199	.78361	.78494	.78705	.78516	.78519	.78523
249	.78457	.78509	.78591	.78519	.78519	.78520
299	.78495	.78515	.78547	.78519	.78519	.78519
349	.78510	.78517	.78530	.78519	.78519	.78519
399	.78515	.78518	.78523	.78519	.78519	.78519
	BICUB τ=1			BICUB τ=10, λ=10		
19	.46282	.82472	.88482	.56274	.79389	.90891
39	.74129	.79484	.83477	.77771	.79672	.82180
59	.76652	.79280	.82624	.78648	.79493	.80796
79	.77538	.79302	.81732	.79084	.79468	.80137
99	.78132	.79342	.81081	.79298	.79481	.79816
119	.78545	.79381	.80619	.79406	.79496	.79661
139	.78834	.79414	.80292	.79460	.79505	.79587
159	.79036	.79440	.80061	.79488	.79511	.79551
179	.79178	.79461	.79899	.79502	.79514	.79534
199	.79278	.79476	.79785	.79510	.79516	.79525
299	.79475	.79509	.79562	.79517	.79517	.79517

Table 3. Detailed intermediate results for example 1 with a 5 × 5 grid, using BICUB with $\tau = 10$, $\lambda = 10$, and starting with $h(\cdot) = 0$

iteration	d^-	H(y)	d^+	iteration	d^-	H(y)	d^+
1	0.00000	.99000	0.99000	.		.	
2	0.00000	.98100	0.98100	.		.	
3	0.00178	.97215	0.97290	.		.	
4	0.00789	.96275	0.97185	132	.79543	.79504	.79596
5	0.01975	.95232	0.97747	133	.79454	.79504	.79595
6	0.02346	.94077	0.98826	134	.79455	.79504	.79594
7	0.06090	.92857	1.00065	135	.79456	.79505	.79592
8	0.08880	.91642	0.96336	136	.79457	.79505	.79591
9	0.12029	.90447	0.95531	137	.79458	.79505	.79590
10		.78633		138	.79459	.79505	.79588
11	-0.61653	.77949	1.52678	139	.79460	.79505	.79587
12	-0.15458	.77725	1.04796	140		.79507	.
13	0.09811	.77794	1.00814	141	.79471	.79507	.79573
14	0.28001	.78084	1.02287	142	.79471	.79508	.79572
15	0.32000	.78381	0.99537	143	.79472	.79508	.79572
16	0.41525	.78674	0.95528	144	.79473	.79508	.79571
17	0.47106	.78940	0.93672	145	.79474	.79508	.79570
18	0.50684	.79172	0.91794	146	.79474	.79508	.79569
19	0.56274	.79389	0.90891	147	.79475	.79508	.79568
20		.81234		148	.79476	.79508	.79567
21	0.53745	.80956	1.14075	149	.79476	.79509	.79566
22	0.63682	.80775	1.11100	150		.79510	
23	0.64938	.80647	1.08539	151	.79484	.79510	.79556
24	0.68398	.80532	1.04246	152	.79484	.79510	.79556
25	0.72688	.80433	1.04133	153	.79485	.79510	.79555
26	0.73479	.80347	0.99842	154	.79486	.79510	.79554
27	0.74647	.80270	0.93582	155	.79486	.79511	.79554
28	0.75029	.80201	0.87887	156	.79486	.79511	.79553
29	0.75299	.80139	0.83857	157	.79487	.79511	.79552
30		.79583		158	.79488	.79511	.79551
31	0.69577	.79610	0.84770	159	.79488	.79511	.79551
32	0.73739	.79641	0.84594	160		.79512	
				161	.79493	.79512	.79544
				162	.79494	.79512	.79544
.		.		.		.	
.		.		.		.	
.		.		.		.	

Using BICUB interpolation with $\lambda = \tau = 10$, performing 400 iterations with a 7 $\times$ 9 grid, $(u_1, \ldots, u_7) = (0, 3, 6, 13, 20, 30, 40)$ and $(v_1, \ldots, v_9) = (0, 3, 6, 13, 20, 30, 40, 50, 60)$, we obtain $H(y) = 0.7134$, $d^- = 0.7123$ and $d^+ = 0.7138$. Refining the grid to 11 $\times$ 15 and performing 100 new iterations yields $H(y) = 0.7135$, and adding another 100 iterations with a 23 $\times$ 33 grid yields again $H(y) = 0.7135$, with $d^- = 0.7134$ and $d^+ = 0.7144$. The value of $H(y)$ has stabilized and a further grid refinement is obviously unnecessary.

5. CONCLUSION

We have seen that value iteration dynamic programming with interpolation can provide precise performance measures for three-stage transfer lines with buffers of virtually any sizes, while state aggregation fails. Over-relaxation steps can also accelerate the convergence. The numerical illustrations given here are representative of a larger amount of numerical testing that was performed. In principle, this approach can be extended to lines with more than 3 stages, and be a good alternative to simulation. For lines with many stages, however, interpolation would be more difficult than for the three-stage case, and the amount of computing will soon become overwhelming. Simulation and decomposition probably remain the sole efficient methods to approximate the performance measures for these cases.

For three-stage lines, the time spent by the Gershwin and Schick's method is $O((N_1 + N_2)^3)$, while for our method, it is $O(\alpha\beta)$ for each iteration. As the number of iterations in our case depends on the precision we need, it is difficult to compare the two methods. For small values of N_1 and N_2, for which the Gershwin and Schick's method can be used, the latter is certainly faster. They needed about 1 min of CPU time on their machine to solve example 1, while we spent about 20 min on our VAX for the same problem. But for large values of N_1 and N_2, our method not only becomes competitive w.r. to CPU time, but still provides reliable results, while the method of Gershwin and Schick fails, due to numerical instabilities.

Acknowledgements—This research has been supported by NSERC-Canada Grant No. A5463 and FCAC-Québec Grant No. EQ2831. The author wish to thank Jacques Malenfant and Marc Veilleux, who contributed to the numerical implementation.

REFERENCES

1. H. Akima, A method of Bivariate interpolation and smooth surface fitting based on local procedures. *Commun. ACM* **17**(1), 18–20 and 26–31 (1974).
2. D. P. Bertsekas, *Dynamic Programming and Stochastic Control*. Academic Press (1976).
3. J. A. Buzacott, Automatic transfer lines with buffer stocks. *Int. J. Prod. Res.* **5**, 183–200 (1967).
4. J. A. Buzacott & L. E. Hanifin, Models of automatic transfer lines with inventory banks—a review and comparison. *AIIE Trans.* **10**, 197–207 (1978).
5. S. B. Gershwin, An efficient decomposition method for the approximate evaluation of production lines with finite storage space, Presentation TB8.2 at ORSA/TIMS 84 in San Francisco (May 1984).
6. S. B. Gershwin and I. C. Schick, Modeling and analysis of three-stage transfer lines with unreliable machines and finite buffers. *Ops. Res.* **31**(2), 354–380 (1983).
7. A. Haurie & P. L'Ecuyer, Approximation and bounds in discrete event dynamic programming, Les cahiers du GERAD (G-83-25), Ecole des H.E.C., Montréal (1983).
8. Y. C. Ho, M. A. Eyler & T. T. Chien, A new approach to determine parameter sensitivities of transfer lines. *Man. Sci.* **29**(6), 700– (1983).
9. J. Masso & M. L. Smith, Interstage storages for three stage lines subject to stochastic failures. *AIIE Trans.* **6**(4), 354–358 (1974).
10. E. L. Porteus & J. C. Totten, Accelerated computation of the expected discounted return in a Markov chain. *Ops. Res.* **26**(2), 350–358 (1978).
11. D. Reetz, Solution of a Markovian decision problem by successive over-relaxation. *Z. Oper. Res.* **17**, 29–32 (1973).
12. D. J. White, Dynamic programming, Markov chains, and the method of successive approximations. *J. Math. Anal. Appl.* **6**, 373–376 (1963).
13. H. Yamashima & K. Okamura, Analysis of in-process buffers for multi-stage transfer line systems. *Int. J. Prod. Res.* **21**(2), 183–195 (1983).