For purposes of illustration, 10 replications of each of the four designs were run, using independent sampling (i.e., different random numbers for all runs). The results are presented in Table 12.6, together with sample means ($\bar{Y}_i$), sample standard deviations (S_i), and sample variances (S_i^2), plus the observed difference of sample means ($\bar{Y}_1 - \bar{Y}_i$) and the standard error (s.e.) of the observed difference. It is observed that all three confidence intervals for $\theta_1 - \theta_i (i = 2, 3, 4)$ contain zero. Therefore, no strong conclusion is possible from these data and this sample size. By contrast, a sample size of ten was sufficient, when using CRN, to provide strong evidence that design 3 is superior to design 1.

Notice the large increase in standard error of the estimated difference with independent sampling versus with common random numbers. These standard errors are compared in Table 12.7. In addition, a careful examination of Tables 12.5 and 12.6 illustrates the superiority of CRN. In Table 12.5, in all 10 replications, system design 3 has a smaller average response time than does system design 1. By comparing replications 1 and 2 in Table 12.5, it can be seen that a random-number stream that leads to high congestion and large response times in system design 1, as in the first replication, produces results of similar magnitude across all four system designs. Similarly, when system design 1 exhibits relatively low congestion and low response times, as in the second replication, all system designs produce relatively low average response times. This similarity of results on each replication is due, of course, to the use of common random numbers across systems. By contrast, for independent sampling, Table 12.6 shows no such similarity across system designs. In only 5 of the 10 replications is the average response time for system design 3 smaller than that for system design 1, although the average difference in response times across all 10 replications is approximately the same magnitude in each case: 5.69 minutes when using CRN, and 5.89 minutes when using independent

Table 12.6 Analysis of Output Data for the Vehicle-Inspection System that Uses Independent Sampling

	Average Response Time for System Design			
Replication, r	*1,* Y_{r1}	*2,* Y_{r2}	*3,* Y_{r3}	*4,* Y_{r4}
1	63.72	59.37	52.00	59.03
2	32.24	50.06	47.04	49.97
3	40.28	60.63	53.21	60.18
4	36.94	46.36	40.88	45.44
5	36.29	68.87	50.85	66.65
6	56.94	66.44	60.42	66.03
7	34.10	27.51	26.70	27.45
8	63.36	47.98	40.12	47.50
9	49.29	29.92	28.59	29.84
10	87.20	47.14	41.62	46.44
Sample mean $\bar{Y}_i$	50.04	50.43	44.14	49.85
S_i	17.70	13.98	10.76	13.64
S_i^2	313.38	195.54	115.74	185.98
$\bar{Y}_1 - \bar{Y}_i$		−0.39	5.89	0.18
s.e.$(\bar{Y}_1 - \bar{Y}_i)$		7.13	6.55	7.07

Table 12.7 Comparison of Standard Errors Arising from CRN with those from Independent Sampling, for the Vehicle-Inspection Problem

Difference in Sample Means	*Standard Error When Using*		*Percentage Increase*
	CRN Sampling	*Independent Sampling*	
$\bar{Y}_1 - \bar{Y}_2$	0.67	7.13	1064%
$\bar{Y}_1 - \bar{Y}_3$	1.69	6.55	388%
$\bar{Y}_1 - \bar{Y}_4$	0.74	7.07	955%

sampling. The greater variability of independent sampling is reflected also in the standard errors of the point estimates: ±1.69 minutes for CRN versus ± 6.55 minutes for independent sampling, an increase of 388%, as seen in Table 12.7. This example illustrates again the advantage of CRN.

As stated previously, CRN does not yield a variance reduction in all simulation models. It is recommended that a pilot study be undertaken and variances estimated to confirm (or possibly deny) the assumption that CRN will reduce the variance (or standard error) of an estimated difference. The reader is referred to the discussion in Section 12.1.3.

Some of the exercises at the end of this chapter provide an opportunity to compare CRN and independent sampling and to compute simultaneous confidence intervals under the Bonferroni approach.

12.2.2 Bonferroni Approach to Selecting the Best

Suppose that there are K system designs, and the ith design has expected performance θ_i. At a gross level, we are interested in which system is best, where "best" is defined to be having maximum expected performance.[1] At a more refined level, we could also be interested in how much better the best is relative to each alternative, because secondary criteria that are not reflected in the performance measure θ_i (such as ease of installation, cost to maintain, etc.) could tempt us to choose an inferior system if it is not deficient by much.

If system design i is the best, then $\theta_i - \max_{j \neq i} \theta_j$ is equal to the difference in performance between the best and the second best. If system design i is not the best, then $\theta_i - \max_{j \neq i} \theta_j$ is equal to the difference between system i and the best. The selection procedure we describe in this section focuses on the parameters $\theta_i - \max_{j \neq i} \theta_j$ for $i = 1, 2, \ldots, K$.

Let i^* denote the (unknown) index of the best system. As a general rule, the smaller the true difference $\theta_{i^*} - \max_{j \neq i^*} \theta_j$ is, and the more certain we want to be that we find the best system, the more replications are required to achieve our goal. Therefore, instead of demanding that we find i^*, we can compromise and ask to find i^* with high probability whenever the difference between system i^* and the others is at least some practically significant amount. More precisely, we want the probability that we select the best system to be at least $1 - \alpha$ whenever $\theta_{i^*} - \max_{j \neq i^*} \theta_j \geq \epsilon$. If there are one or more systems that are within ϵ of the best, then we will be satisfied to select either the best or any one of the near best. Both the probability of correct selection, $1 - \alpha$, and the practically significant difference, ϵ, will be under our control.

The following procedure achieves the desired probability of correct selection (Nelson and Matejcik [1995]). And because we are also interested in how much each system differs from the best, it also forms $100(1 - \alpha)\%$ confidence intervals for $\theta_{i^*} - \max_{j \neq i^*} \theta_j$ for $i = 1, 2, \ldots, K$. The procedure is valid for normally distributed data when either CRN or independent sampling is being used.

[1]If "best" is defined to be having minimum expected performance, then the procedure in this section is easily modified, as we illustrate in the example.

Two-Stage Bonferroni Procedure

1. Specify the practically significant difference ϵ, the probability of correct selection $1 - \alpha$, and the first-stage sample size $R_0 \geq 10$. Let $t = t_{\alpha/(K-1),R_0-1}$.
2. Make R_0 replications of system i to obtain $Y_{1i}, Y_{2i}, \ldots, Y_{R_0,i}$, for systems $i = 1, 2, \ldots, K$.
3. Calculate the first-stage sample means $\bar{Y}_i$, $i = 1, 2, \ldots, K$. For all $i \neq j$, calculate the sample variance of the difference,[2]

$$S_{ij}^2 = \frac{1}{R_0 - 1}\sum_{r=1}^{R_0}\left(Y_{ri} - Y_{rj} - (\bar{Y}_i - \bar{Y}_j)\right)^2$$

Let $\hat{S}^2 = \max_{i \neq j} S_{ij}^2$, the largest sample variance.
4. Calculate the second-stage sample size,

$$R = \max\left\{R_0, \left\lceil \frac{t^2\hat{S}^2}{\epsilon^2} \right\rceil\right\}$$

where $\lceil \cdot \rceil$ means to round up.
5. Make $R - R_0$ additional replications of system i to obtain the output data $Y_{R_0+1,i}, Y_{R_0+2,i}$, for $i = 1, 2, \ldots, K$.[3]
6. Calculate the overall sample means

$$\bar{\bar{Y}}_i = \frac{1}{R}\sum_{r=1}^{R} Y_{ri}$$

for $i = 1, 2, \ldots, K$.
7. Select the system with largest $\bar{\bar{Y}}_i$ as the best.

Also form the confidence intervals

$$\min\{0, \bar{\bar{Y}}_i - \max_{j \neq i} \bar{\bar{Y}}_j - \epsilon\} \leq \theta_i - \max_{j \neq i} \theta_j \leq \max\{0, \bar{\bar{Y}}_i - \max_{j \neq i} \bar{\bar{Y}}_j + \epsilon\}$$

for $i = 1, 2, \ldots, K$.

The confidence intervals in Step 7 are not like the usual $\pm$ intervals presented elsewhere in this chapter. Perhaps the most useful interpretation of them is as follows. Let $\hat{i}$ be the index of the system selected as best. Then, for each of the other systems i, we make one of the declarations:

- If $\bar{\bar{Y}}_i - \bar{\bar{Y}}_{\hat{i}} + \epsilon \leq 0$, then declare system i to be inferior to the best.
- If $\bar{\bar{Y}}_i - \bar{\bar{Y}}_{\hat{i}} + \epsilon > 0$, then declare system i to be statistically indistinguishable from the best (and, therefore, system i might be the best).

Example 12.4: Continued

Recall that, in Example 12.4, we considered $K = 4$ different designs for the vehicle-inspection station. Suppose that we would like 0.95% confidence of selecting the best (smallest expected response time) system design when

[2]Notice that S_{ij}^2 is algebraically equivalent to S_D^2, the sample variance of $D_r = Y_{ri} - Y_{rj}$, for $r = 1, 2, \ldots, R_0$.

[3]If it is more convenient, a total of R replications can be generated from system i by restarting the entire experiment.

the best differs from the second best by at least two minutes. This is a minimization problem, so we focus on the differences $\theta_i - \min_{j \neq i} \theta_j$ for $i = 1, 2, 3, 4$. Then we can apply the Two-Stage Bonferroni Procedure as follows:

1. $\epsilon = 2$ minutes, $1 - \alpha = 0.95$, $R_0 = 10$, and $t = t_{0.0167,9} = 2.508$.
2. The data in Table 12.5, which was obtained by using CRN, is employed.
3. From Table 12.5, we get $S_{12}^2 = S_{D_2}^2 = 4.498$, $S_{13}^2 = S_{D_3}^2 = 28.498$, and $S_{14}^2 = S_{D_4}^2 = 5.489$. By similar calcultions, we obtain $S_{23}^2 = 11.857$, $S_{24}^2 = 0.119$, and $S_{34}^2 = 9.849$.
4. Since $\hat{S}^2 = S_{13}^2 = 28.498$ is the largest sample variance,

$$R = \max\left\{10, \left\lceil \frac{(2.508)^2(28.498)}{2^2} \right\rceil\right\} = \max\{10, \lceil 44.8 \rceil\} = 45$$

5. Make $45 - 10 = 35$ additional replications of each system.
6. Calculate the overall sample means

$$\bar{\bar{Y}}_i = \frac{1}{45}\sum_{r=1}^{45} Y_{ri}$$

   for $i = 1, 2, 3, 4$.
7. Select the system with smallest $\bar{\bar{Y}}_i$ as the best.
   Also, form the confidence intervals

$$\min\{0, \bar{\bar{Y}}_i - \min_{j \neq i} \bar{\bar{Y}}_j - 2\} \le \theta_i - \min_{j \neq i} \theta_j \le \max\{0, \bar{\bar{Y}}_i - \min_{j \neq i} \bar{\bar{Y}}_j + 2\}$$

   for $i = 1, 2, 3, 4$.

12.2.3 Bonferroni Approach to Screening

When a two-stage procedure is not possible, or when there are many systems, it could be useful to divide the set of systems into those that could be the best and those that can be eliminated from further consideration. For this purpose, a screening or *subset selection* procedure is useful. The following procedure, due to Nelson *et al.* [2001], guarantees that the retained subset contains the true best system with probability $\ge 1 - \alpha$ when the data are normally distributed and either independent sampling or CRN is used. The subset may contain all K of the systems, only one system, or some number in between, depending on the number of replications and the sample means and sample variances.

Screening Procedure

1. Specify the probability of correct selection $1 - \alpha$ and common sample size from each system, $R \ge 2$. Let $t = t_{\alpha/(K-1),R-1}$.
2. Make R replications of system i to obtain $Y_{1i}, Y_{2i}, \ldots, Y_{Ri}$ for systems $i = 1, 2, \ldots, K$.
3. Calculate the sample means $\bar{Y}_{\cdot i}$ for $i = 1, 2, \ldots, K$. For all $i \neq j$, calculate the sample variance of the difference,

$$S_{ij}^2 = \frac{1}{R-1}\sum_{r=1}^{R}(Y_{ri} - Y_{rj} - (\bar{Y}_{\cdot i} - \bar{Y}_{\cdot j}))^2$$

4. If bigger is better, then retain system i in the selected subset if

$$\bar{Y}_{\cdot i} \geq \bar{Y}_{\cdot j} - t\frac{S_{ij}}{\sqrt{R}} \text{ for all } j \neq i$$

If smaller is better, then retain system i in the selected subset if

$$\bar{Y}_{\cdot i} \leq \bar{Y}_{\cdot j} + t\frac{S_{ij}}{\sqrt{R}} \text{ for all } j \neq i$$

All system designs that are not retained can be eliminated from further consideration.

Example 12.4: Continued

Suppose we want to see whether any of the designs for the vehicle-inspection station can be eliminated on the basis of only the 10 replications in Table 12.5. Summaries of the sample means and variances of the differences are as follows:

$\bar{Y}_{\cdot i}$	1	2	3	4
	50.04	49.24	44.35	48.78

S_{ij}^2	1	2	3	4
1		4.498	28.498	5.489
2			11.857	0.119
3				9.84

The appropriate critical value to obtain 95% confidence that the selected subset contains the true best is $t = t_{0.0167,9} = 2.508$. Recall that smaller response time is better. Applying the Subset Selection Procedure, system designs 1, 2, and 4 can all be eliminated, because

$$\bar{Y}_1 = 50.04 \not\leq \bar{Y}_3 + t\sqrt{\frac{S_{13}^2}{R}} = 44.35 + 2.508\sqrt{\frac{28.498}{10}} = 48.58$$

$$\bar{Y}_2 = 49.24 \not\leq \bar{Y}_3 + t\sqrt{\frac{S_{23}^2}{R}} = 44.35 + 2.508\sqrt{\frac{11.857}{10}} = 47.08$$

$$\bar{Y}_4 = 48.78 \not\leq \bar{Y}_3 + t\sqrt{\frac{S_{43}^2}{R}} = 44.35 + 2.508\sqrt{\frac{9.84}{10}} = 46.84$$

Thus, in this case there was adequate data to select the best, system design 3, with 95% confidence. Had more than one system survived the subset selection, then we could perform additional analysis on that subset, perhaps using the Two-Stage Bonferroni Procedure.

12.3 METAMODELING

Suppose that there is a simulation output response variable, Y, that is related to k independent variables, say $x_1, x_2, \ldots, x_k$. The dependent variable, Y, is a random variable, while the independent variables $x_1, x_2, \ldots, x_k$ are called design variables and are usually subject to control. The true relationship between the variables Y and x is represented by the (often complex) simulation model. Our goal is to approximate this relationship by a simpler mathematical function called a metamodel. In some cases, the analyst will know the exact form of the functional relationship between Y and $x_1, x_2, \ldots, x_k$, say $Y = f(x_1, x_2, \ldots, x_k)$. However, in most cases, the functional relationship is unknown, and the analyst must select an appropriate f containing unknown parameters, and then estimate those parameters from a set of data (Y, x). Regression analysis is one method for estimating the parameters.

Example 12.5

An insurance company promises to process all claims it receives each day by the end of the next day. It has developed a simulation model of its proposed claims-processing system to evaluate how hard it will be to meet this promise. The actual number and types of claims that will need to be processed each day will vary, and the number may grow over time. Therefore, the company would like to have a model that predicts the total processing time as a function of the number of claims received.

The primary value of a metamodel is to make it easy to answer "what if" questions, such as, what the processing time will be if there are x claims. Evaluating a function f, or perhaps its derivatives, at a number of values of x is typically much easier than running a simulation experiment for each value.

12.3.1 Simple Linear Regression

Suppose that it is desired to estimate the relationship between a single independent variable x and a dependent variable Y, and suppose that the true relationship between Y and x is suspected to be linear. Mathematically, the expected value of Y for a given value of x is assumed to be

$$E(Y \mid x) = \beta_0 + \beta_1 x \tag{12.21}$$

where β_0 is the intercept on the Y axis, an unknown constant; and β_1 is the slope, or change in Y for a unit change in x, also an unknown constant. It is further assumed that each observation of Y can be described by the model

$$Y = \beta_0 + \beta_1 x + \epsilon \tag{12.22}$$

where ϵ is a random error with mean zero and constant variance σ^2. The regression model given by Equation (12.22) involves a single variable x and is commonly called a simple linear regression model.

Suppose that there are n pairs of observations (Y_1, x_1), (Y_2, x_2), $\ldots, (Y_n, x_n)$. These observations can be used to estimate β_0 and β_1 in Equation (12.22). The method of least squares is commonly used to form the estimates. In the method of least squares, β_0 and β_1 are estimated in such a way that the sum of the squares of the deviations between the observations and the regression line is minimized. The individual observations in Equation (12.22) can be written as

$$Y_i = \beta_0 + \beta_1 x_i + \epsilon_i,\ i = 1, 2, \ldots, n \tag{12.23}$$

where $\epsilon_1, \epsilon_2, \ldots$ are assumed to be uncorrelated random variables.

Each ϵ_i in Equation (12.23) is given by

$$\epsilon_i = Y_i - \beta_0 - \beta_1 x_i \tag{12.24}$$

and represents the difference between the observed response, Y_i, and the expected response, $\beta_0 + \beta_1 x_i$, predicted by the model in Equation (12.21). Figure 12.3 shows how ϵ_i is related to x_i,Y_i, and $E(Y_i|x_i)$.

The sum of squares of the deviations given in Equation (12.24) is given by

$$L = \sum_{i=1}^{n} \epsilon_i^2 = \sum_{i=1}^{n} (Y_i - \beta_0 - \beta_1 x_i)^2 \tag{12.25}$$

and L is called the least-squares function. It is convenient to rewrite Y_i as follows:

$$Y_i = \beta_0' + \beta_1(x_i - \bar{x}) + \epsilon_i \tag{12.26}$$

where $\beta_0' = \beta_0 + \beta_1 \bar{x}$ and $\bar{x} = \sum_{i=1}^{n} x_i / n$. Equation (12.26) is often called the transformed linear regression model. Using Equation (12.26), Equation (12.25) becomes

$$L = \sum_{i=1}^{n} [Y_i - \beta_0' - \beta_1(x_i - \bar{x})]^2$$

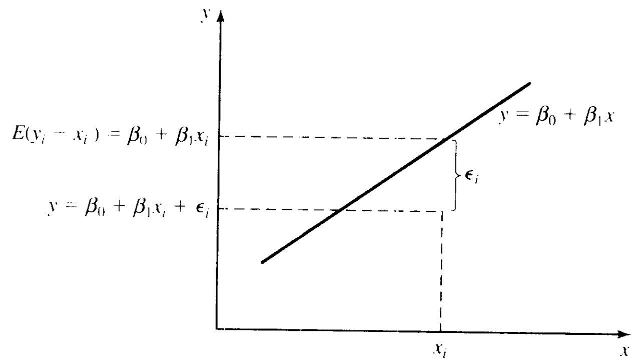
To minimize L, find $\partial L / \partial \beta_0'$ and $\partial L / \partial \beta_1$, set each to zero, and solve for $\hat{\beta}_0'$ and $\hat{\beta}_1$. Taking the partial derivatives and setting each to zero yields

$$n\hat{\beta}_0' = \sum_{i=1}^{n} Y_i$$

$$\hat{\beta}_1 \sum_{i=1}^{n} (x_i - \bar{x})^2 = \sum_{i=1}^{n} Y_i(x_i - \bar{x}) \tag{12.27}$$

Equations (12.27) are often called the "normal equations," which have the solutions

$$\hat{\beta}_0' = \bar{Y} = \sum_{i=1}^{n} \frac{Y_i}{n} \tag{12.28}$$



y
$E(y_i - x_i) = \beta_0 + \beta_1 x_i$
$y = \beta_0 + \beta_1 x_i + \epsilon_i$
$y = \beta_0 + \beta_1 x$
ϵ_i
x_i
x

Figure 12.3 Relationship of ϵ_i to x_i, Y_i, and $E(Y_i|x_i)$.

and

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} Y_i(x_i - \bar{x})}{\sum_{i=1}^{n} (x_i - \bar{x})^2} \qquad (12.29)$$

The numerator in Equation (12.29) is rewritten for computational purposes as

$$S_{xy} = \sum_{i=1}^{n} Y_i(x_i - \bar{x}) = \sum_{i=1}^{n} x_i Y_i - \frac{\left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} Y_i\right)}{n} \qquad (12.30)$$

where S_{xy} denotes the corrected sum of cross products of x and Y. The denominator of Equation (12.29) is rewritten for computational purposes as

$$S_{xx} = \sum_{i=1}^{n} (x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n} \qquad (12.31)$$

where S_{xx} denotes the corrected sum of squares of x. The value of $\hat{\beta}_0$ can be retrieved easily as

$$\hat{\beta}_0 = \hat{\beta}_0' - \hat{\beta}_1 \bar{x} \qquad (12.32)$$

Example 12.6: Calculating $\hat{\beta}_0$ and $\hat{\beta}_1$

The simulation model of the claims-processing system in Example 12.5 was executed with initial conditions x = 100, 150, 200, 250, and 300 claims received the previous day. Three replications were obtained at each setting. The response Y is the number of hours required to process x claims. The results are shown in Table 12.8. The graphical relationship between the number of claims received and total processing time is shown in

Table 12.8 Simulation Results for Processing Time Given x Claims

Number of Claims x	*Hours of Processing Time Y*
100	8.1
100	7.8
100	7.0
150	9.6
150	8.5
150	9.0
200	10.9
200	13.3
200	11.6
250	12.7
250	14.5
250	14.7
300	16.5
300	17.5
300	16.3

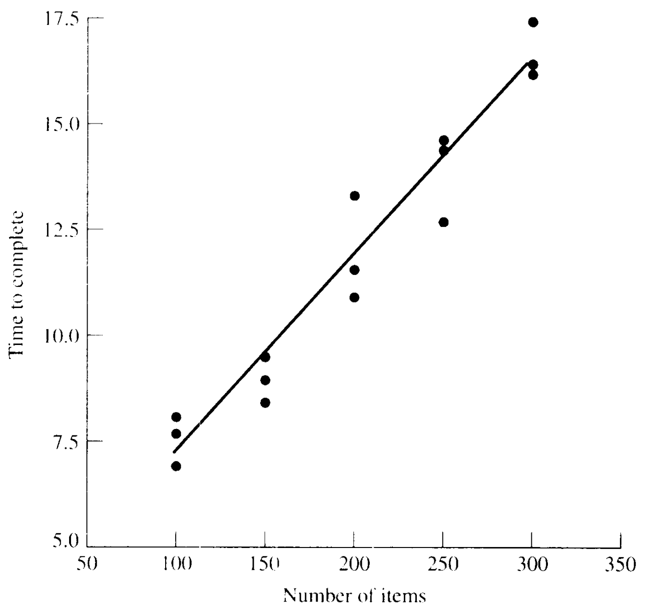


Figure 12.4 Relationship between number of claims and hours of processing time.

Figure 12.4. Such a display is called a scatter diagram. Examination of this scatter diagram indicates that there is a strong relationship between number of claims and processing time. The tentative assumption of the linear model given by Equation (12.22) appears to be reasonable.

With the processing times as the Y_i values (the dependent variables) and the number of claims as the x_i values (the independent variables), $\hat{\beta}_0$ and $\hat{\beta}_1$ can be found by the following computations: $n = 15$, $\sum_{i=1}^{15} x_i = 3000$, $\sum_{i=1}^{15} Y_i = 178$, $\sum_{i=1}^{15} x_i^2 = 675{,}000$, $\sum_{i=1}^{15} x_i Y_i = 39080$, and $\bar{x} = 3000/15 = 200$.

From Equation (12.30) S_{xy} is calculated as

$$S_{xy} = 39{,}080 - \frac{(3000)(178)}{15} = 3480$$

From Equation (12.31), S_{xx} is calculated as

$$S_{xx} = 675{,}000 - \frac{(3000)^2}{15} = 75{,}000$$

Then, $\hat{\beta}_1$ is calculated from Equation (12.29) as

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}} = \frac{3480}{75{,}000} = 0.0464$$

As shown in Equation (12.28), $\hat{\beta}_0'$ is just $\bar{Y}$, or

$$\hat{\beta}_0' = \frac{178}{15} \approx 11.8667$$

To express the model in the original terms, compute $\hat{\beta}_0$ from Equation (12.32) as

$$\hat{\beta}_0 = 11.8667 - 0.0464(200) = 2.5867$$

Then an estimate of the mean of Y given x, $E(Y|x)$, is given by

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x = 2.5867 + 0.0464x \tag{12.33}$$

For a given number of claims, x, this model can be used to predict the number of hours required to process them. The coefficient $\hat{\beta}_1$ has the interpretation that each additional claim received adds an expected 0.0464 hours, or 2.8 minutes, to the expected total processing time.

Regression analysis is widely used and frequently misused. Several of the common abuses are briefly mentioned here. Relationships derived in the manner of Equation (12.33) are valid for values of the independent variable within the range of the original data. The linear relationship that has been tentatively assumed may not be valid outside the original range. In fact, we know from queueing theory that mean processing time may increase rapidly as the number of claims approaches the capacity of the system. Therefore, Equation (12.33) can be considered valid only for $100 \le x \le 300$. Regression models are not advised for extrapolation purposes.

Care should be taken in selecting variables that have a plausible causal relationship with each other. It is quite possible to develop statistical relationships that are unrelated in a practical sense. For example, an attempt might be made to relate monthly output of a steel mill to the weight of reports appearing on a manager's desk during the month. A straight line may appear to provide a good model for the data, but the relationship between the two variables is tenuous. A strong observed relationship does not imply that a causal relationship exists between the variables. Causality can be inferred only when analysis uncovers some plausible reasons for its existence. In Example 12.5 it is reasonable that starting with more claims implies that more time is needed to process them. Therefore, a relationship of the form of Equation (12.33) is at least plausible.

12.3.2 Testing for Significance of Regression

In Section 12.3.1, it was assumed that a linear relationship existed between Y and x. In Example 12.5, a scatter diagram, shown in Figure 12.4, relating number of claims and processing time was prepared to evaluate whether a linear model was a reasonable tentative assumption prior to the calculation of $\hat{\beta}_0$ and $\hat{\beta}_1$. However, the adequacy of the simple linear relationship should be tested prior to using the model for predicting the response, Y_i, given an independent variable, x_i. There are several tests which may be conducted to aid in determining model adequacy. Testing whether the order of the model tentatively assumed is correct, commonly called the "lack-of-fit test," is suggested. The procedure is explained by Box and Draper [1987], Hines, Montgomery, Goldsman, and Borror [2002], and Montgomery [2000].

Testing for the significance of regression provides another means for assessing the adequacy of the model. The hypothesis test described next requires the additional assumption that the error component ϵ_i is normally distributed. Thus, the complete assumptions are that the errors are NID(0, σ^2)—that is, normally and independently distributed with mean zero and constant variance σ^2. The adequacy of the assumptions can and should be checked by residual analysis, discussed by Box and Draper [1987], Hines, Montgomery, Goldsman, and Borror [2002], and Montgomery [2000].

Testing for significance of regression is one of many hypothesis tests that can be developed from the variance properties of $\hat{\beta}_0$ and $\hat{\beta}_1$. The interested reader is referred to the references just cited for extensive discussion of hypothesis testing in regression. Just the highlights of testing for significance of regression are given in this section.

Suppose that the alternative hypotheses are

$H_0 : \beta_1 = 0$
$H_1 : \beta_1 \neq 0$

Failure to reject H_0 indicates that there is no linear relationship between x and Y. This situation is illustrated in Figure 12.5. Notice that two possibilities exist. In Figure 12.5(a), the implication is that x is of little value in explaining the variability in Y, and that $\hat{y} = \bar{Y}$ is the best estimator. In Figure 12.5(b), the implication is that the true relationship is not linear.

Alternatively, if H_0 is rejected, the implication is that x is of value in explaining the variability in Y. This situation is illustrated in Figure 12.6. Here, also, two possibilities exist. In Figure 12.6(a), the straight-line model is adequate. However, in Figure 12.6(b), even though there is a linear effect of x, a model with higher-order terms (such as x^2, x^3, ...) is necessary. Thus, even though there may be significance of regression, testing of the residuals and testing for lack of fit are needed to confirm the adequacy of the model.

The appropriate test statistic for significance of regression is given by

$$t_0 = \frac{\hat{\beta}_1}{\sqrt{MS_E / S_{xx}}} \tag{12.34}$$

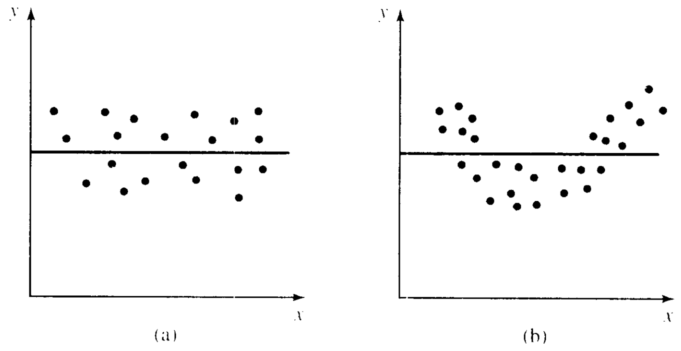


Figure 12.5 Failure to reject $H_0 : \beta_1 = 0$.

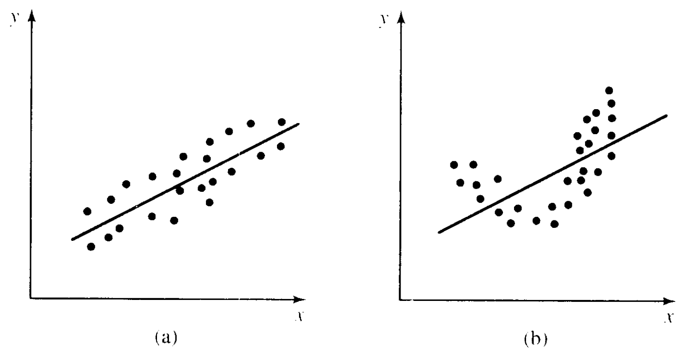


Figure 12.6 $H_0 : \beta_1 = 0$ is rejected.

where MS_E is the mean squared error. The error is the difference between the observed value, Y_i, and the predicted value, $\hat{y}_i$, at x_i, or $e_i = Y_i - \hat{y}_i$. The squared error is given by $\sum_{i=1}^{n} e_i^2$, and the mean squared error, given by

$$MS_E = \sum_{i=1}^{n} \frac{e_i^2}{n-2} \tag{12.35}$$

is an unbiased estimator of $\sigma^2 = V(\epsilon_i)$. The direct method can be used to calculate $\sum_{i=1}^{n} e_i^2$: Calculate each $\hat{y}_i$, compute e_i^2, and sum all the e_i^2 values, $i = 1, 2, \ldots, n$. However, it can be shown that

$$\sum_{i=1}^{n} e_i^2 = S_{yy} - \hat{\beta}_1 S_{xy} \tag{12.36}$$

where S_{yy}, the corrected sum of squares of Y, is given by

$$S_{yy} = \sum_{i=1}^{n} Y_i^2 - \frac{\left(\sum_{i=1}^{n} Y_i\right)^2}{n} \tag{12.37}$$

and S_{xy} is given by Equation (12.30). Equation (12.36) could be easier to use than the direct method.

The statistic defined by Equation (12.34) has the t distribution with $n - 2$ degrees of freedom. The null hypothesis H_0 is rejected if $|t_0| > t_{\alpha/2,n-2}$.

Example 12.7: Testing for Significance of Regression

Given the results in Example 12.6, the test for the significance of regression is conducted. One more computation is needed prior to conducting the test. That is, $\sum_{i=1}^{n} Y_i^2 = 2282.94$. Using Equation (12.37) yields

$$S_{yy} = 2282.94 - \frac{(178)^2}{15} = 170.6734$$

Then $\sum_{i=1}^{15} e_i^2$ is computed according to Equation (12.36) as

$$\sum_{i=1}^{15} e_i^2 = 170.6734 - 0.0464(3480) = 9.2014$$

Now, the value of MS_E is calculated from Equation (12.35):

$$MS_E = \frac{9.2014}{13} = 0.7078$$

The value of t_0 can be calculated by using Equation (12.34) as

$$t_0 = \frac{0.0464}{\sqrt{0.7078/75000}} = 15.13$$

Since $t_{0.025,13} = 2.16$ from Table A.5, we reject the hypothesis that $\beta_1 = 0$. Thus, there is significant evidence that x and Y are related.

12.3.3 Multiple Linear Regression

If the simple linear regression model of Section 12.3.1 is inadequate, several other possibilities exist. There could be several independent variables, so that the relationship is of the form

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m + \epsilon \tag{12.38}$$

Notice that this model is still linear, but has more than one independent variable. Regression models having the form shown in Equation (12.38) are called multiple linear regression models. Another possibility is that the model is of a quadratic form such as

$$Y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon \tag{12.39}$$

Equation (12.39) is also a linear model which may be transformed to the form of Equation (12.38) by letting $x_1 = x$ and $x_2 = x^2$.

Yet another possibility is a model of the form such as

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \epsilon$$

which is also a linear model. The analysis of these three models with the forms just shown, and related models, can be found in Box and Draper [1987], Hines, Montgomery, Goldsman, and Borror [2002], Montgomery [2000], and other applied statistics texts; and also in Kleijnen [1987, 1998], which is concerned primarily with the application of these models in simulation.

12.3.4 Random-Number Assignment for Regression

The assignment of random-number seeds or streams is part of the design of a simulation experiment.[4] Assigning a different seed or stream to different design points (settings for $x_1, x_2, \ldots x_m$ in a multiple linear regression) guarantees that the responses Y from different design points will be statistically independent. Similarly, assigning the same seed or stream to different design points induces dependence among the corresponding responses, by virtue of their all having the same source of randomness.

Many textbook experimental designs assume independent responses across design points. To conform to this assumption, we must assign different seeds or streams to each design point. However, it is often useful to assign the same random number seeds or streams to all of the design points—in other words, to use common random numbers.

The intuition behind common random numbers for metamodels is that a fairer comparison among design points is achieved if the design points are subjected to the same experimental conditions, specifically the same source of randomness. The mathematical justification is as follows: Suppose we fit the simple linear regression $Y_i = \beta_0 + \beta_1 x_i + \epsilon_i$ and obtain least squares estimates $\hat{\beta}_0$ and $\hat{\beta}_1$. Then an estimator of the expected difference in performance between design points i and j is

$$\hat{\beta}_0 + \hat{\beta}_1 x_i - (\hat{\beta}_0 + \hat{\beta}_1 x_j) = \hat{\beta}_1 (x_i - x_j)$$

when x_i and x_j are fixed design points, $\hat{\beta}_1$ determines the estimated difference between design points i and j, or for that matter between any other two values of x. Therefore, common random numbers can be expected to reduce the variance of $\hat{\beta}_1$ and, more generally, reduce the variance of all of the slope terms in a multiple linear regression. Common random numbers typically do not reduce the variance of the intercept term, $\hat{\beta}_0$.

[4]This section is based on Nelson [1992].

The least-squares estimators $\hat{\beta}_0$ and $\hat{\beta}_1$ are appropriate regardless of whether we use common random numbers, but the associated statistical analysis is affected by that choice. For statistical analysis of a metamodel under common random numbers, see Kleijnen [1988] and Nelson [1992].

12.4 OPTIMIZATION VIA SIMULATION

Consider the following examples.[5]

Example 12.8: Materials Handling System (MHS)

Engineers need to design a MHS consisting of a large automated storage and retrieval device, automated guided vehicles (AGVs), AGV stations, lifters, and conveyors. Among the design variables they can control are the number of AGVs, the load per AGV, and the routing algorithm used to dispatch the AGVs. Alternative designs will be evaluated according to AGV utilization, transportation delay for material that needs to be moved, and overall investment and operation costs.

Example 12.9: Liquified Natural Gas (LNG) Transportation

A LNG transportation system will consist of LNG tankers and of loading, unloading, and storage facilities. In order to minimize cost, designers can control tanker size, number of tankers in use, number of jetties at the loading and unloading facilities, and capacity of the storage tanks.

Example 12.10: Automobile Engine Assembly

In an assembly line, a large buffer (queue) between workstations could increase station utilization—because there will tend to be something waiting to be processed—but drive up space requirements and work-in-process inventory. An allocation of buffer capacity that minimizes the sum of these competing costs is desired.

Example 12.11: Traffic Signal Sequencing

Civil engineers want to sequence the traffic signals along a busy section of road to reduce driver delay and the congestion occurring along narrow cross streets. For each traffic signal, the length of the red, green, and green-turn-arrow cycles can be set individually.

Example 12.12: On-Line Services

A company offering on-line information services over the Internet is changing its computer architecture from central mainframe computers to distributed workstation computing. The numbers and types of CPUs, the network structure, and the allocation of processing tasks all need to be chosen. Response time to customer queries is the key performance measure.

What do these design problems have in common? Clearly, a simulation model could be useful in each, and all have an implied goal of finding the best design relative to some performance measures (cost, delay, etc.). In each example, there are potentially a very large number of alternative designs, ranging from tens to thousands, and certainly more than the 2 to 10 we considered in Section 12.2.2. Some of the examples contain a diverse collection of decision variables: discrete (number of AGVs, number of CPUs), continuous (tanker size, red-cycle length) and qualitative (routing strategy, algorithm for allocating processing tasks). This makes developing a metamodel, as described in Section 12.3, difficult.

All of these problems fall under the general topic of "optimization via simulation," where the goal is to minimize or maximize some measures of system performance and system performance can be evaluated only by running a computer simulation. Optimization via simulation is a relatively new, but already vast, topic, and commercial software has become widely available. In this section, we describe the key issues that should be considered in undertaking optimization via simulation, provide some pointers to the available literature, and give one example algorithm.

[5]Some of these descriptions are based on Boesel, Nelson, and Ishii [2003].

12.4.1 What Does 'Optimization via Simulation' Mean?

Optimization is a key tool used by operations researchers and management scientists, and there are well-developed algorithms for many classes of problems, the most famous being linear programming. Much of the work on optimization deals with problems in which all aspects of the system are treated as being known with certainty; most critically, the performance of any design (cost, profit, makespan, etc.) can be evaluated exactly.

In stochastic, discrete-event simulation, the result of any simulation run is a random variable. For notation, let $x_1, x_2, \ldots, x_m$ be the m controllable design variables and let $Y(x_1, x_2, \ldots, x_m)$ be the observed simulation output performance on one run. To be concrete, x_1, x_2, x_3 might denote the number of AGVs, the load per AGV, and the routing algorithm used to dispatch the AGVs, respectively, in Example 12.8, while $Y(x_1, x_2, x_3)$ could be total MHS acquisition and operation cost.

What does it mean to "optimize" $Y(x_1, x_2, \ldots, x_m)$ with respect to $x_1, x_2, \ldots, x_m$? Y is a random variable, so we cannot optimize the *actual* value of Y. The most common definition of optimization is

$$\text{maximize or minimize } E(Y(x_1, x_2, \ldots, x_m)) \tag{12.40}$$

In other words, the mathematical expectation, or long-run average, of performance is maximized or minimized. *This is the default definition of optimization used in all commercial packages of which we are aware.* In our example, $E(Y(x_1, x_2, x_3))$ is the expected, or long-run average cost of operating the MHS with x_1 AGVs, x_2 load per AGV, and routing algorithm x_3.

It is important to note that (12.40) is not the only possible definition, however. For instance, we might want to select the MHS design that has the best chance of costing less than \$$D$ to purchase and operate, changing the objective to

$$\text{maximize } \Pr\,(Y(x_1, x_2, x_3) \leq D)$$

We can fit this objective into formulation (12.40) by defining a new performance measure

$$Y'(x_1, x_2, x_3) = \begin{cases} 1, & \text{if } Y(x_1, x_2, x_3) \leq D \\ 0, & \text{otherwise} \end{cases}$$

and maximizing $E(Y'(x_1, x_2, x_3))$ instead.

A more complex optimization problem occurs when we want to select the system design that is *most likely* to be the best. Such an objective is relevant when one-shot, rather than long-run average, performance matters. Examples include a Space Shuttle launch, or the delivery of a unique, large order of products. Bechhofer, Santner, and Goldsman [1995] address this problem under the topic of "multinomial selection."

We have been assuming that a system design $x_1, x_2, \ldots, x_m$ can be evaluated in terms of a single performance measure, Y, such as cost. Obviously, this may not always be the case. In the MHS example, we might also be interested in some measure of system productivity, such as throughput or cycle time. At present, multiple objective optimization via simulation is not well developed. Therefore, one of three strategies is typically employed:

1. Combine all of the performance measures into a single measure, the most common being cost. For instance, the revenue generated by each completed product in the MHS could represent productivity and be included as a negative cost.
2. Optimize with respect to one key performance measure, but then evaluate the top solutions with respect to secondary performance measures. For instance, the MHS could be optimized with respect to expected cost, and then the cycle time could be compared for the top 5 designs. *This approach requires that information on more than just the best solution be maintained.*

3. Optimize with respect to one key performance measure, but consider only those alternatives that meet certain constraints on the other performance measures. For instance, the MHS could be optimized with respect to expected cost for those alternatives whose expected cycle time is less than a given threshold.

12.4.2 Why is Optimization via Simulation Difficult?

Even when there is no uncertainty, optimization can be very difficult if the number of design variables is large, the problem contains a diverse collection of design variable types, and little is known about the structure of the performance function. Optimization via simulation adds an additional complication: The performance of a particular design cannot be evaluated exactly, but instead must be estimated. Because we have estimates, it is not possible to conclude with assurance that one design is better than another, and this uncertainty frustrates optimization algorithms that try to move in improving directions. In principle, one can eliminate this complication by making so many replications, or such long runs, at each design point that the performance estimate has essentially no variance. In practice, this could mean that very few alternative designs will be explored, because of the time required to simulate each one.

The existence of sampling variability forces optimization via simulation to make compromises. The following are the standard ones:

- **Guarantee a prespecified probability of correct selection.** The Two-Stage Bonferroni Procedure in Section 12.2.2 is an example of this approach, which allows the analyst to specify the desired chance of being right. Such algorithms typically require either that every possible design be simulated or that a strong functional relationship among the designs (such as a metamodel) apply. Other algorithms can be found in Goldsman and Nelson [1998].
- **Guarantee asymptotic convergence.** There are many algorithms that guarantee convergence to the global optimal solution as the simulation effort (number of replications, length of replications) becomes infinite. These guarantees are useful because they indicate that the algorithm tends to get to where the analyst wants it to go. However, convergence can be slow, and there is often no guarantee as to how good the reported solution is when the algorithm is terminated in finite time (as it must be in practice). See Andradóttir [1998] for specific algorithms that apply to discrete- or continuous-variable problems.
- **Optimal for deterministic counterpart.** The idea here is to use an algorithm that would find the optimal solution *if the performance of each design could be evaluated with certainty*. An example might be applying a standard nonlinear programming algorithm to the simulation optimization problem. It is typically up to the analyst to make sure that enough simulation effort is expended (replications or run length) to insure that such an algorithm is not misled by sampling variability. Direct application of an algorithm that assumes deterministic evaluation to a stochastic simulation is not recommended.
- **Robust heuristics.** Many heuristics have been developed for deterministic optimization problems that do not guarantee finding the optimal solution, but nevertheless been shown to be very effective on difficult, practical problems. Some of these heuristics use randomness as part of their search strategy, so one might argue that they are less sensitive to sampling variability than other types of algorithms. Nevertheless, it is still important to make sure that enough simulation effort is expended (replications or run length) to insure that such an algorithm is not misled by sampling variability.

Robust heuristics are the most common algorithms found in commercial optimization via simulation software. We provide some guidance on their use in the next section. See Fu [2002] for a comprehensive discussion of optimization theory versus practice.

12.4.3 Using Robust Heuristics

By a "robust heuristic" we mean a procedure that does not depend on strong problem structure—such as continuity or convexity of $E(Y(x_1, \ldots, x_m))$—to be effective, can be applied to problems with mixed types of decision variables, and—ideally—is tolerant of some sampling variability. Genetic algorithms (GA) and tabu search (TS) are two prominent examples, but there are many others and many variations of them. Such heuristics form the core of most commercial implementations. To give a sense of these heuristics, we describe GA and TS next. We caution the reader that only a high-level description of the simplest version of each procedure is provided. The commercial implementations are much more sophisticated.

Suppose that there are k possible solutions to the optimization via simulation problem. Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$ denote the solutions, where the ith solution $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{im})$ provides specific settings for the m decision variables. The simulation output at solution $\mathbf{x}_i$ is denoted $Y(\mathbf{x}_i)$; this could be the output of a single replication, or the average of several replications. Our goal is to find the solution $\mathbf{x}^*$ that minimizes $E(Y(\mathbf{x}))$.

On each iteration (known as a "generation"), a GA operates on a "population" of p solutions. Denote the population of solutions on the jth iteration as $\mathbf{P}(j) = \{\mathbf{x}_1(j), \mathbf{x}_2(j), \ldots, \mathbf{x}_p(j)\}$. There may be multiple copies of the same solution in $\mathbf{P}(j)$, and $\mathbf{P}(j)$ may contain solutions that were discovered on previous iterations. From iteration to iteration, this population evolves in such a way that good solutions tend to survive and give birth to new, and hopefully better, solutions, while inferior solutions tend to be removed from the population. The basic GA is given here:

Basic GA

Step 1. Set the iteration counter $j = 0$, and select (perhaps randomly) an initial population of p solutions $\mathbf{P}(0) = \{\mathbf{x}_1(0), \ldots, \mathbf{x}_p(0)\}$.

Step 2. Run simulation experiments to obtain performance estimates $Y(\mathbf{x})$ for all p solutions $\mathbf{x}(j)$ in $\mathbf{P}(j)$.

Step 3. Select a population of p solutions from those in $\mathbf{P}(j)$ in such a way that those with smaller $Y(\mathbf{x})$ values are more likely, but not certain, to be selected. Denote this population of solutions as $\mathbf{P}(j + 1)$.

Step 4. Recombine the solutions in $\mathbf{P}(j + 1)$ via crossover (which joins parts of two solutions $\mathbf{x}_i(j + 1)$ and $\mathbf{x}_i(j + 1)$ to form a new solution) and mutation (which randomly changes a part of a solution $\mathbf{x}_i(j + 1)$).

Step 5. Set $j = j + 1$ and go to Step 2.

The GA can be terminated after a specified number of iterations, when little or no improvement is noted in the population, or when the population contains p copies of the same solution. At termination, the solution $\mathbf{x}^*$ that has the smallest $Y(\mathbf{x})$ value in the last population is chosen as best (or alternatively, the solution with the smallest $Y(\mathbf{x})$ over all iterations could be chosen).

GAs are applicable to almost any optimization problem, because the operations of selection, crossover, and mutation can be defined in a very generic way that does not depend on specifics of the problem. However, when these operations are not tuned to the specific problem, a GA's progress can be very slow. Commercial versions are often self-tuning, meaning that they update selection, crossover, and mutation parameters during the course of the search. There is some evidence that GAs are tolerant of sampling variability in $Y(\mathbf{x})$ because they maintain a population of solutions rather than focusing on improving a current-best solution. In other words, it is not critical that the GA rank the solutions in a population of solutions perfectly, because the next iteration depends on the entire population, not on a single solution.

TS, on the other hand, identifies a current best solution on each iteration and then tries to improve it. Improvements occur by changing the solution via "moves." For example, the solution (x_1, x_2, x_3) could be changed to the solution $(x_1 + 1, x_2, x_3)$ by the move of adding 1 to the first decision variable (perhaps x_1 represents the number of AGVs in Example 12.8, so the move would add one more AGV). The "neighbors" of solution $\mathbf{x}$ are all of those solutions that can be reached by legal moves. TS finds the best neighbor solution and moves to it. However, to avoid making moves that return the search to a previously visited solution, moves may become "tabu" (not usable) for some number of iterations. Conceptually, think about how you would find your way through a maze: If you took a path that lead to a dead end, then you would avoid taking that path again (it would be tabu).

The basic TS algorithm is given next. The description is based on Glover [1989].

Basic TS

Step 1. Set the iteration counter $j = 0$ and the list of tabu moves to empty. Select an initial solution $\mathbf{x}^*$ in $\mathbf{X}$ (perhaps randomly).

Step 2. Find the solution $\mathbf{x}'$ that minimizes $Y(\mathbf{x})$ over all of the neighbors of $\mathbf{x}^*$ that are not reached by tabu moves, running whatever simulations are needed to do the optimization.

Step 3. If $Y(\mathbf{x}') < Y(\mathbf{x}^*)$, then $x^* = x'$ (move the current best solution to x').

Step 4. Update the list of tabu moves and go to Step 2.

The TS can be terminated when a specified number of iterations have been completed, when some number of iterations has passed without changing $\mathbf{x}^*$, or when there are no more feasible moves. At termination, the solution $\mathbf{x}^*$ is chosen as best.

TS is fundamentally a discrete-decision-variable optimizator, but continuous decision variables can be discretized, as described in Section 12.4.4. TS aggressively pursues improving solutions, and therefore tends to make rapid progress. However, it is more sensitive to random variability in $Y(\mathbf{x})$, because $\mathbf{x}^*$ is taken to be the *true* best solution so far and attempts are made to improve it. There are probabilistic versions of TS that should be less sensitive, however. An important feature of commercial implementations of TS, which is not present in the Basic TS, is a mechanism for overiding the tabu list when doing so is advantageous.

Next, we offer two suggestions for using commercial products that employ a GA, TS, or other robust heuristic controlling sampling variability, and *restarting*.

Control Sampling Variability

In many cases, it will up to the user to determine how much sampling (replications or run length) will be undertaken at each potential solution. This is a difficult problem in general. Ideally, sampling should increase as the heuristic closes in on the better solutions, simply because it is much more difficult to distinguish solutions that are close in expected performance from those that differ widely. Early in the search, it may be easy for the heuristic to identify good solutions and search directions, because clearly inferior solutions are being compared to much better ones, but late in the search this might not be the case.

If the analyst must specify a fixed number of replications per solution that will be used through the search, then a preliminary experiment should be conducted. Simulate several designs, some at the extremes of the solution space and some nearer the center. Compare the apparent best and apparent worst of these designs, using the approaches in Section 12.1. Using the technique described in Section 12.1.4, find the minimum for the number of replications required to declare these designs to be statistically significantly different. This is the minimum number of replications that should be used.

After the optimization run has completed, perform a second set of experiments on the top 5 to 10 designs identified by the heuristic. Use the comparison techniques in Section 12.2–12.2.3 to rigorously evaluate which are the best or near-best of these designs.

Restarting

Because robust heuristics provide no guarantees that they converge to the optimal solution for optimization via simulation, it makes sense to run the optimization two or more times to see which run yields the best solution. Each optimization run should use different random number seeds or streams and, ideally, should start from different initial solutions. Try starting the optimization at solutions on the extremes of the solution space, in the center of the space and at randomly generated solutions. If people familiar with the system suspect that certain designs will be good, be sure to include them as possible starting solutions for the heuristic.

12.4.4 An Illustration: Random Search

In this section, we present an algorithm for optimization via simulation known as random search. The specific implementation is based on Algorithm 2 in Andradóttir [1998], which provides guaranteed asymptotic convergence under certain conditions. Thus, it will find the true optimal solution if permitted to run long enough. However, in practice, convergence can be slow, and the memory requirements of this particular version of random search can be quite large. Even though random search is not a "robust heuristic," we will also use it to demonstrate some strategies we would employ in conjunction with such heuristics and to demonstrate why optimization via simulation is tricky even with what appears to be an uncomplicated algorithm.

The random-search algorithm that we present requires that there be a finite number of possible system designs (although that number may be quite large). This might seem to rule out problems with continuous decision variables, such as conveyor speed. In practice, however, apparently continuous decision variables can often be discretized in a reasonable way. For instance, if conveyor speed can be anything from 60 to 120 feet per minute, little may be lost by treating the possible conveyor speeds as 60, 61, 62, ..., 120 feet per minute (61 possible values). Note, however, that there are algorithms designed specifically for continuous-variable problems (Andradóttir [1998]).

Again, let the k possible solutions to the optimization via simulation problem be denoted $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$, where the ith solution $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{im})$ provides specific settings for the m decision variables. The simulation output at solution $\mathbf{x}_i$ is denoted $Y(\mathbf{x}_i)$; this could be the output of a single replication or the average of several replications. Our goal is to find the solution $\mathbf{x}^*$ that minimizes $E(Y(\mathbf{x}))$.

On each iteration of the random-search algorithm, we compare a current good solution to a randomly chosen competitor. If the competitor is better, then it becomes the current good solution. When we terminate the search, the solution we choose is the one that has been visited most often (which means that we expect to revisit solutions many times).

Random-Search Algorithm

Step 1. Initialize counter variables $C(i) = 0$ for $i = 1, 2, \ldots, k$. Select an initial solution i^0, and set $C(i^0) = 1$. ($C(i)$ counts the number of times we visit solution i.)

Step 2. Choose another solution i' from the set of all solutions *except* i^0 in such a way that each solution has an equal chance of being selected.

Step 3. Run simulation experiments at the two solutions i^0 and i' to obtain outputs $Y(i^0)$ and $Y(i')$. If $Y(i') < Y(i^0)$, then set $i^0 = i'$. (See note following Step 4.)

Step 4. Set $C(i^0) = C(i^0) + 1$. If not done, then go to Step 2. If done, then select as the estimated optimal solution xi^* such that $C(i^*)$ is the largest count.

Note that, if the problem is a maximization problem, then replace Step 3 with

Step 3. Run simulation experiments at the two solutions i^0 and i' to obtain outputs $Y(i^0)$ and $Y(i')$. If $Y(i') > Y(i^0)$, then set $i^0 = i'$.

One of the difficult problems with many optimization-via-simulation algorithms is knowing when to stop. (Exceptions include algorithms that guarantee a probability of correct selection.) Typical rules might be to stop after a certain number of iterations, stop when the best solution has not changed much in several iterations, or stop when all time available to solve the problem has been exhausted. Whatever rule is used, we recommend applying a statistical selection procedure, such as the Two-Stage Bonferroni Procedure in Section 12.2.2, to the 5 to 10 apparently best solutions. This is done to evaluate which among them is the true best with guaranteed confidence. If the raw data from the search have been saved, then these data can be used as the first-stage sample for a two-stage selection procedure (Boesel, Nelson, and Ishii [2003]).

Example 12.13: Implementing Random Search

Suppose that a manufacturing system consists of 4 stations in series. The zeroth station always has raw material available. When the zeroth station completes work on a part, it passes the part along to the first station, then the first passes the part to the second, and so on. Buffer space between stations 0 and 1, 1 and 2, and 2 and 3 is limited to 50 parts total. If, say, station 2 finishes a part but there is no buffer space available in front of station 3, then station 2 is blocked, meaning that it cannot do any further work. The question is how to allocate these 50 spaces to minimize the expected cycle time per part over one shift.

Let x_i be the number of buffer spaces in front of station i. Then the decision variables are x_1, x_2, x_3 with the constraint that $x_1 + x_2 + x_3 = 50$ (it makes no sense to allocate fewer buffer spaces than we have available). This implies a total of 1326 possible designs (can you figure out how this number is computed?).

To simplify the presentation of the random-search algorithm, let the counter for solution (x_1, x_2, x_3) be denoted as $C(x_1, x_2, x_3)$.

Random Search Algorithm

Step 1. Initialize 1326 counter variables $C(x_1, x_2, x_3) = 0$, one for each of the possible solutions (x_1, x_2, x_3). Select an initial solution, say $(x_1 = 20, x_2 = 15, x_3 = 15)$ and set $C(20, 15, 15) = 1$.

Step 2. Choose another solution from the set of all solutions *except* (20, 15, 15) in such a way that each solution has an equal chance of being selected. Suppose (11, 35, 4) is chosen.

Step 3. Run simulation experiments at the two solutions to obtain estimates of the expected cycle time $Y(20, 15, 15)$ and $Y(11, 35, 4)$. Suppose that $Y(20, 15, 15) < Y(11, 35, 4)$. Then (20, 15, 15) remains as the current good solution.

Step 4. Set $C(20, 15, 15) = C(20, 15, 15) + 1$.

Step 2. Choose another solution from the set of all solutions *except* (20, 15, 15) in such a way that each solution has an equal chance of being selected. Suppose (28, 12, 10) is chosen.

Step 3. Run simulation experiments at the two solutions to obtain estimates of the expected cycle time $Y(20, 15, 15)$ and $Y(28, 12, 10)$. Suppose that $Y(28, 12, 10) < Y(20, 15, 15)$. Then (28, 12, 10) becomes the current good solution.

Step 4. Set $C(28, 12, 10) = C(28, 12, 10) + 1$.

Step 2. Choose another solution from the set of all solutions *except* (28, 12, 10) in such a way that each solution has an equal chance of being selected. Suppose (0, 14, 36) is chosen.

Step 3. Continue...

When the search is terminated, we select the solution (x_1, x_2, x_3) that gives the largest $C(x_1, x_2, x_3)$ count. As we discussed earlier, the top 5 to 10 solutions should then be subjected to a separate statistical analysis to determine which among them is the true best (with high confidence). In this case, the solutions with the largest counts would receive the second analysis.

Despite the apparent simplicity of the Random-Search Algorithm, we have glossed over a subtle issue that often arises in algorithms with provable performance. In Step 2, the algorithm must randomly choose a solution such that all are equally likely to be selected (except the current one). How can this be accomplished in Example 12.13? The constraint that $x_1 + x_2 + x_3 = 50$ means that x_1, x_2 and x_3 cannot be sampled independently. One might be tempted to sample x_1 as a discrete uniform random variable on 0 to 50, then sample x_2 as a discrete uniform on 0 to $50 - x_1$, and finally set $x_3 = 50 - x_1 - x_2$. But this method does not make all solutions equally likely, as the following illustration shows: Suppose that x_1 is randomly sampled to be 50. Then the trial solution must be (50, 0, 0); there is only one choice. But if $x_1 = 49$, then both (49, 1, 0) and (49, 0, 1) are possible. Thus, $x_1 = 49$ should be more likely than $x_1 = 50$ if all solutions with $x_1 + x_2 + x_3 = 50$ are to be equally likely.

12.5 SUMMARY

This chapter provided a basic introduction to the comparative evaluation of alternative system designs based on data collected from simulation runs. It was assumed that a fixed set of alternative system designs had been selected for consideration. Comparisons based on confidence intervals and the use of common random numbers were emphasized. A brief introduction to metamodels—whose purpose is to describe the relationship between design variables and the output response—and to optimization via simulation—whose purpose is to select the best from among a large and diverse collection of system designs—was also provided. There are many additional topics of potential interest (beyond the scope of this text) in the realm of statistical analysis techniques relevant to simulation. Some of these topics are

1. experimental design models, whose purpose is to discover which factors have a significant impact on the performance of system alternatives;
2. output-analysis methods other than the methods of replication and batch means;
3. variance-reduction techniques, which are methods to improve the statistical efficiency of simulation experiments (common random numbers being an important example).

The reader is referred to Banks [1998] and Law and Kelton [2000] for discussions of these topics and of others relevant to simulation.

The most important idea in Chapters 11 and 12 is that simulation output data require a statistical analysis in order to be interpreted correctly. In particular, a statistical analysis can provide a measure of the precision of the results produced by a simulation and can provide techniques for achieving a specified precision.

REFERENCES

ANDRADÓTTIR, S. [1998], "Simulation Optimization," Chapter 9 in *Handbook of Simulation*, J. Banks, ed., Wiley, New York.

BANKS, J., ed. [1998], *Handbook of Simulation*, Wiley, New York.

BECHHOFER, R. E., T. J. SANTNER, AND D. GOLDSMAN [1995], *Design and Analysis for Statistical Selection, Screening and Multiple Comparisons*, Wiley, New York.

BOESEL, J., B. L. NELSON, AND N. ISHII [2003], "A Framework for Simulation-Optimization Software," *IIE Transactions*, Vol. 35, pp. 221–229.

BOX, G. E. P., AND N. R. DRAPER [1987], *Empirical Model-Building and Response Surfaces*, Wiley, New York.

FU, M. C. [2002], "Optimization for Simulation: Theory vs. Practice," *INFORMS Journal on Computing*, Vol. 14, pp. 192–215.

GLOVER, F. [1989], "Tabu Search—Part I," *ORSA Journal on Computing*, Vol. 1, pp. 190–206.

GOLDSMAN, D., AND B. L. NELSON [1998], "Comparing Systems via Simulation," Chapter 8 in *Handbook of Simulation*, J. Banks, ed., Wiley, New York.

HINES, W. W., D. C. MONTGOMERY, D. M. GOLDSMAN, AND C. M. BORROR [2002], *Probability and Statistics in Engineering*, 4th ed., Wiley, New York.
HOCHBERG Y., AND A. C. TAMHANE [1987], *Multiple Comparison Procedures*, Wiley, New York.
HSU, J. C. [1996], *Multiple Comparisons: Theory and Methods*, Chapman & Hall, New York.
KLEIJNEN, J. P. C. [1975], *Statistical Techniques in Simulation, Parts I and II*, Dekker, New York.
KLEIJNEN, J. P. C. [1987], *Statistical Tools for Simulation Practitioners*, Dekker, New York.
KLEIJNEN, J. P. C. [1988], "Analyzing Simulation Experiments with Common Random Numbers," *Management Science*, Vol. 34, pp. 65–74.
KLEIJNEN, J. P. C. [1998], "Experimental Design for Sensitivity Analysis, Optimization, and Validation of Simulation Models," Chapter 6 in *Handbook of Simulation*, J. Banks, ed., Wiley, New York.
LAW, A. M., AND W. D. KELTON [2000], *Simulation Modeling and Analysis*, 3d ed., McGraw-Hill, New York.
MONTGOMERY, D. C. [2000], *Design and Analysis of Experiments*, 5th ed., Wiley, New York.
NELSON, B. L., AND F. J. MATEJCIK [1995], "Using Common Random Numbers for Indifference-Zone Selection and Multiple Comparisons in Simulation," *Management Science*, Vol. 41, pp. 1935–1945.
NELSON, B. L., J. SWANN, D. GOLDSMAN, AND W.-M. T. SONG [2001], "Simple Procedures for Selecting the Best System when the Number of Alternatives is Large," *Operations Research*, Vol. 49, pp. 950–963.
NELSON, B. L. [1987], "Some Properties of Simulation Interval Estimators Under Dependence Induction," *Operations Research Letters*, Vol. 6, pp. 169–176.
NELSON, B. L. [1992], "Designing Efficient Simulation Experiments," *1992 Winter Simulation Conference Proceedings*, pp. 126–132.
WRIGHT, R. D., AND T. E. RAMSAY, JR. [1979], "On the Effectiveness of Common Random Numbers," *Management Science*, Vol. 25, pp. 649–656.

EXERCISES

1. Reconsider the dump-truck problem of Example 3.5, which was also analyzed in Example 12.2. As business expands, the company buys new trucks, making the total number of trucks now equal to 16. The company desires to have a sufficient number of loaders and scales so that the average number of trucks waiting at the loader queue plus the average number at the weigh queue is no more than three. Investigate the following combinations of number of loaders and number of scales:

Number of Scales	*Number of Loaders* 2	3	4
1	–	–	–
2	–	–	–

The loaders being considered are the "slow" loaders in Example 12.2. Loading time, weighing time, and travel time for each truck are as previously defined in Example 12.2. Use common random numbers to the greatest extent possible when comparing alternative systems designs. The goal is to find the smallest number of loaders and scales to meet the company's objective of an average total queue length of no more than three trucks. In your solution, take into account the initialization conditions, run length, and number of replications needed to achieve a reasonable likelihood of valid conclusions.

2. In Exercise 11.5, consider the following alternative (M, L) policies:

Investigate the relative costs of these policies, using suitable modifications of the simulation model developed in Exercise 11.5. Compare the four system designs on the basis of long-run mean monthly cost. First make four replications of each (M, L) policy, using common random numbers to the greatest

			L	
			Low 30	*High* 40
M	Low	50	(50, 30)	(50, 40)
	High	100	(100, 30)	(100, 40)

extent possible. Each replication should have a 12-month initialization phase followed by a 100-month data-collection phase. Compute confidence intervals having an overall confidence level of 90% for mean monthly cost for each policy. Then estimate the additional replications needed to achieve confidence intervals that do not overlap. Draw conclusions as to which is the best policy.

3. Reconsider Exercise 11.6. Compare the four inventory policies studied in Exercise 2, taking the cost of rush orders into account when computing monthly cost.

4. In Exercise 11.8, investigate the effect of the order quantity on long-run mean daily cost. Each order arrives on a pallet on a delivery truck, so the permissible order quantities, Q, are multiples of 10 (i.e., Q may equal 10, or 20, or 30, ...). In Exercise 11.8, the policy $Q = 20$ was investigated.

(a) First, investigate the two policies $Q = 10$ and $Q = 50$. Use the run lengths, and so on, suggested in Exercise 11.8. On the basis of these runs, decide whether the optimal Q, say Q^*, is between 10 and 50 or is greater than 50. (The cost curve as a function of Q should have what kind of shape?)

(b) Using the results in part (a), suggest two additional values for Q and simulate the two policies. Draw conclusions. Include an analysis of the strength of your conclusions.

5. In Exercise 11.10, find the number of cards Q that the card shop owner should purchase to maximize the profit with an error of approximately \$5.00. Use the following expression to generate Q value

$$Q = 300 \pm 100$$

For each run, generate a uniform random variate to get the Q value and for that Q value compute profit.

6. In Exercise 11.10, investigate the effect of target level M and review period N on mean monthly cost. Consider two target levels, M, determined by ± 10 from the target level used in Exercise 11.10, and consider review periods N of 1 month and 3 months. Which (N, M) pair is best, according to these simulations?

7. Reconsider Exercises 11.12 and 11.13, which involved the scheduling rules (or queue disciplines) first-in–first-out (FIFO) and priority-by-type (PR) in a job shop. In addition to these two rules, consider a shortest imminent operation (SIO) scheduling rule. For a given station, all jobs of the type with the smallest mean processing time are given highest priority. For example, when using an SIO rule at station 1, jobs are processed in the following order: type 2 first, then type 1, and type 3 last. Two jobs of the same type are processed on a FIFO basis. Develop a simulation experiment to compare the FIFO, PR, and SIO rules on the basis of mean total response time over all jobs.

8. In Exercise 11.12 (the job shop with FIFO rule), find the minimum number of workers needed at each station to avoid bottlenecks. A bottleneck occurs when average queue lengths at a station increase steadily over time. (Do not confuse increasing average queue length due to an inadequate number of servers with increasing average queue length due to initialization bias. In the former case, average queue length continues to increase indefinitely and server utilization is 1.0. In the latter case, average queue length eventually levels off and server utilization is less than 1.) Report on utilization of workers and total time it takes for a job to get through the job shop, by type and over all types. (*Hint*: If server utilization at a work station is 1.0, and if average queue length tends to increase linearly as simulation run length increases, it is a good possibility that the work station is unstable and therefore is a bottleneck. In this case, at least one additional worker is needed at the work station. Use queueing theory, namely $\lambda/c_i\mu < 1$, to suggest the minimum number of workers needed at station 1. Recall that λ is the arrival rate, $1/\mu$ is the overall mean service time for one job with one worker, and c_i is the number of workers at station i. Attempt to use the same basic condition, $\lambda/c_i\mu < 1$, to suggest an initial number of servers at station i for $i = 2, 3, 4$.)

9. **(a)** Repeat Exercise 8 for the PR scheduling rule (see Exercise 11.13).
(b) Repeat Exercise 8 for the SIO scheduling rule (see Exercise 12.7).
(c) Compare the minimum required number of workers for each scheduling rule: FIFO, versus PR, versus SIO.

10. With the minimum number of workers found in Exercises 9 and 10 for the job shop of Exercise 11.12, consider adding one worker to the entire shop. This worker can be trained to handle the processing at only one station. At which station should this worker be placed? How does this additional worker affect mean total response time over all jobs? Over type 1 jobs? Investigate the job shop with and without the additional worker for each scheduling rule: FIFO, PR, SIO.

11. In Exercise 11.16, suppose that a buffer of capacity one item is constructed in front of each worker. Design an experiment to investigate whether this change in system design has a significant impact upon individual worker utilizations (ρ_1, ρ_2, ρ_3 and ρ_4). At the very least, compute confidence intervals for $\rho_1^0 - \rho_1^1$ and $\rho_4^0 - \rho_4^1$, where ρ_i^s, is utilization for worker i when the buffer has capacity s.

12. A clerk in the admissions office at Small State University processes requests for admissions materials. The time to process requests depends on the program of interest (e.g., industrial engineering, management science, computer science, etc.) and on the level of the program (Bachelors, Masters, Ph.D.). Suppose that the processing time is modeled well as normally distributed, with mean 7 minutes and standard deviation 2 minutes. At the beginning of the day it takes the clerk some time to get set to begin working on requests; suppose that this time is modeled well as exponentially distributed, with mean 20 minutes. The admissions office typically receives between 40 and 60 requests per day.

Let x be the number of applications received on a day, and let Y be the time required to process them (including the set-up time). Fit a metamodel for $E(Y|x)$ by making n replications at the design points $x = 40, 50, 60$. Notice that, in this case, we know that the correct model is

$$E(Y|x) = \beta_0 + \beta_1 x = 20 + 7x$$

(Why?) Begin with $n = 2$ replications at each design point and estimate β_0 and β_1. Gradually increase the number of replications and observe how many are required for the estimates to be close to the true values.

13. Repeat the previous exercise using CRN. How do the results change?

14. The usual statistical analysis used to test for $\beta_1 \neq 0$ does not hold if we use CRN. Where does it break down?

15. Riches and Associates retains its cash reserves primarily in the form of certificates of deposit (CDs), which earn interest at an annual rate of 8%. Periodically, however, withdrawals must be made from these CDs in order to pay suppliers, etc. These cash outflows are made through a checking account that earns no interest. The need for cash cannot be predicted with certainty. Transfers from CDs to checking can be made instantaneously, but there is a "substantial penalty" for early withdrawal from CDs. Therefore, it might make sense for R&A to make use of the overdraft protection on their checking account, which charges interest at a rate of \$0.00033 per dollar per day (i.e., 12% per year) for overdrafts.

R&A likes simple policies in which it transfers a fixed amount, a fixed number of times, per year. Currently, it makes 6 transfers per year, of \$18,250 each time. Your job is to find a policy that reduces its long-run cost per day.

Judging from historical patterns, demands for cash arrive a rate of about 1 per day, with the arrivals being modeled well as a Poisson process. The amount of cash needed to satisfy each demand is reasonably represented by a lognormally distributed random variable with mean \$300 and standard deviation \$150.

The penalty for early withdrawal is different for different CDs. It averages \$150 for each withdrawal (regardless of size), but the actual penalty can be modeled as a uniformly distributed random variable with range \$100 to \$200.

Use cash level in checking to determine the length of the initialization phase. Make enough replications that your confidence interval for the difference in long-run cost per day does not contain zero. Be sure to use CRN in your experiment design.

16. If you have access to commercial optimization-via-simulation software, test how well it works as the variability of the simulation outputs increases. Use a simple model, such as $Y = x^2 + \varepsilon$, where ε is a random variable with a $N(0, \sigma^2)$ distribution, and for which the optimal solution is known ($x = 0$ for minimization, in this case). See how quickly, or whether, the software can find the true optimal solution as σ^2 increases. Next, try more complex models with more than one design variable.

17. For Example 12.12, show why there are 1326 solutions. Then derive a way to sample x_1, x_2, and x_3 such that $x_1 + x_2 + x_3 = 50$ and all outcomes are equally likely.

18. A critical electronic component with mean time to failure of x years can be purchased for $2x$ thousand dollars (thus, the more reliable the component, the more expensive it is). The value of x is restricted to being between 1 to 10 years, and the actual time to failure is modeled as exponentially distributed. The mission for which the component is to be used lasts one year; if the component fails in less than one year, then there is a cost of \$20,000 for early failure. What value of x should be chosen to minimize the expected total cost (purchase plus early failure)?

To solve this problem, develop a simulation that generates a total cost for a component with mean time to failure of x years. This requires sampling an exponentially distributed random variable with mean x, and then computing the total cost as $2000x$ plus 20,000 if the failure time is less than 1. Fit a quadratic metamodel in x and use it to find the value of x that minimizes the fitted model. [Hints: Select several values of x between 1 and 10 as design points. At each value of x, let the response variable $Y(x)$ be the average of at least 30 observations of total cost.]

19. The demand for an item follows $N(10, 2)$. It is required to avoid the shortage. Let Q be the order quantity. Assuming Q to be an integer between 10 and 150, determine the optimal value for Q that maximizes the probability, so that the shortage is equal to zero. Use random search algorithm.

20. If you have access, use any optimization via simulation software to solve Exercise 19.

21. Explore the possibility of applying metaheuristics to search for near-optimal solution using simulation models.

Part V

Applications

13

Simulation of Manufacturing and Material-Handling Systems

Manufacturing and material-handling systems provide one of the most important applications of simulation. Simulation has been used successfully as an aid in the design of new production facilities, warehouses, and distribution centers. It has also been used to evaluate suggested improvements to existing systems. Engineers and analysts using simulation have found it valuable for evaluating the impact of capital investments in equipment and physical facility and of proposed changes to material handling and layout. They have also found it useful to evaluate staffing and operating rules and proposed rules and algorithms to be incorporated into production control systems, warehouse-management control software, and material-handling controls. Managers have found simulation useful in providing a "test drive" before making capital investments, without disrupting the existing system with untried changes.

Section 13.1 provides an introduction and discusses some of the features of simulation models of manufacturing and material-handling systems. Section 13.2 discussed the goals of manufacturing simulation and the most common measures of system performance. Section 13.3 discusses a number of the issues common to many manufacturing and material-handling simulations, including the treatment of downtimes and failure, and trace-driven simulations using actual historical data or historical order files. Section 13.4 provides brief abstracts of a number of reported simulation projects, with references for additional reading. Section 13.5 gives an extended example of a simulation of a small production line, emphasizing the experimentation and analysis of system performance to achieve a desired throughput. For an overview of simulation software for manufacturing and material-handling applications, see Section 4.7.

13.1 MANUFACTURING AND MATERIAL-HANDLING SIMULATIONS

As do all modeling projects, manufacturing and material-handling simulation projects need to address the issues of scope and level of detail. Consider scope as analagous to breadth and level of detail as analagous to depth. Scope describes the boundaries of the project: what's in the model, and what's not. For a subsystem, process, machine, or other component, the project scope determines whether the object is in the model. Then, once a component or subsystem is treated as part of a model, often it can be simulated at many different levels of detail.

The proper scope and level of detail should be determined by the objectives of the study and the questions being asked. On the other hand, level of detail could be constrained by the availability of input data and the knowledge of how system components work. For new, nonexistent systems, data availability might be limited, and system knowledge might be based on assumptions.

Some general guidelines can be provided, but the judgment of experienced simulation analysts working with the customer to define, early in the project, the questions the model is being designed to address provides the most effective basis for selecting a proper scope and a proper level of detail.

Should the model simulate each conveyor section or vehicle movement, or can some be replaced by a simple time delay? Should the model simulate auxiliary parts, or the handling of purchased parts, or can the model assume that such parts are always available at the right location when needed for assembly?

At what level of detail does the control system need to be simulated? Many modern manufacturing facilities, distribution centers, baggage-handling systems, and other material-handling systems are computer controlled by a management-control software system. The algorithms built into such control software play a key role in system performance. Simulation is often used to evaluate and compare the effectiveness of competing control schemes and to evaluate suggested improvements. It can be used to debug and fine-tune the logic of a control system before it is installed.

These questions are representative of the issues that need to be addressed in choosing the correct level of model detail and scope of a project. In turn, the scope and level of model detail limit the type of questions that can be addressed by the model. In addition, models can be developed in an iterative fashion, adding detail for peripheral operations at later stages if such operations are later judged to affect the main operation significantly. It is good advice to start as simple as possible and add detail only as needed.

13.1.1 Models of Manufacturing Systems

Models of manufacturing systems might have to take into account a number of characteristics of such systems, some of which are the following:

- Physical layout
- Labor
  - Shift schedules
  - Job duties and certification
- Equipment
  - Rates and capacities
  - Breakdowns
    - Time to failure
    - Time to repair
    - Resources needed for repair
- Maintenance
  - PM schedule
  - Time and resources required
  - Tooling and fixtures

- Workcenters
  - Processing
  - Assembly
  - Disassembly
- Product
  - Product flow, routing, and resources needed
  - Bill of materials
- Production schedules
  - Made-to-stock
  - Made-to-order
    - Customer orders
    - Line items and quantities
- Production control
  - Assignment of jobs to work areas
  - Task selection at workcenters
  - Routing decisions
- Supplies
  - Ordering
  - Receipt and storage
  - Delivery to workcenters
- Storage
  - Supplies
  - Spare parts
  - Work-in-process (WIP)
  - Finished goods
- Packing and shipping
  - Order consolidation
  - Paperwork
  - Loading of trailers

13.1.2 Models of Material Handling Systems

In manufacturing systems, it is not unusual for 80 to 85% of an item's total time in system to be expended on material handling or on waiting for material handling to occur. This work-in-process (WIP) represents a vast investment, and reductions in WIP and associated delays can result in large cost savings. Therefore, for some studies, detailed material-handling simulations are cost effective.

In some production lines, the material-handling system is an essential component. For example, automotive paint shops typically consist of a power-and-free conveyor system that transports automobile bodies or body parts through the paint booths.

In warehouses, distribution centers, and flow-through and cross-docking operations, material handling is clearly a key component of any material-flow model. Manual warehouses typically use manual fork trucks to move pallets from receiving dock to storage and from storage to shipping dock. More automated distribution centers might use extensive conveyor systems to support putaway, order picking, order sortation, and consolidation.

Models of material-handling systems often have to contain some of the following types of subsystems:

- Conveyors
  - Accumulating
  - Nonaccumulating

- Indexing and other special purpose
- Fixed window or random spacing
- Power and free

Transporters
- Unconstrained vehicles (e.g., manually guided fork trucks)
- Guided vehicles (automated or operator controlled, wire guided chemical paths, rail guided)
- Bridge cranes and other overhead lifts

Storage systems
- Pallet storage
- Case storage
- Small-part storage (totes)
- Oversize items
- Rack storage or block stacked
- Automated storage and retrieval systems (AS/RS) with storage–retrieval machines (SRM)

13.1.3 Some Common Material-Handling Equipment

There are numerous types of material-handling devices common to manufacturing, warehousing, and distribution operations. They include unconstrained transporters, such as carts, manually driven fork-lift trucks, and pallet jacks; guided path transporters, such as AGVs (automated guided vehicles); and fixed-path devices, such as various types of conveyor.

The class of unconstrained transporters, sometimes called free-path transporters, includes carts, fork-lift trucks, pallet jacks, and other manually driven vehicles that are free to travel throughout a facility unconstrained by a guide path of any kind. Unconstrained transporters are not constrained to a network of paths and may choose an alternate path or move around an obstruction. In contrast, the guided-path transporters move along a fixed path, such as chemical trails on the floor, wires imbedded in the floor, or infrared lights placed strategically, or by self-guidance, using radio communications, laser guidance and dead reckoning, and rail. Guided-path transporters sometimes contend with each other for space along their paths and usually have limited options upon meeting obstacles and congestion. Examples of guided-path transporters include the automated guided vehicle (AGV); a rail-guided turret truck for storage and retrievals of pallets in rack storage; and a crane in an AS/RS (automated storage and retrieval system).

The conveyor is a fixed-path device for moving entities from point to point, following a fixed path with specific load, stopping or processing points, and unload points. A conveyor system can consist of numerous connected sections with merges and diverts. Each section can be of one of a number of different types. Examples of conveyor types include belt, powered and gravity roller, bucket, chain, tilt tray, and power-and-free, each with its own characteristics that must be modeled accurately.

Most conveyor sections can be classified as either accumulating or nonaccumulating. An accumulating conveyor section runs continuously. If the forward progress of an item is halted while on the accumulating conveyor, slippage occurs, allowing the item to remain stationary and items behind it to continue moving until they reach the stationary item. Some belt and most roller conveyors operate in this manner. Only items that will not be damaged by bumping into each other can be placed on an accumulating conveyor.

In contrast, after an item is on a nonaccumulating conveyor section, its spacing relative to other items does not change. If one item stops moving, the entire section stops moving, and hence all items on the section stop. For example, nonaccumulating conveyor is used for moving televisions not yet in cartons, for they must be kept at a safe distance from each other while moving from one assembly or testing station to the next. Bucket conveyors, tilt-tray conveyors, some belt conveyors, and conveyors designed to carry heavy loads (usually, pallets) are nonaccumulating conveyors.

Conveyors can also be classified as fixed-window or random spacing. In fixed-window spacing, items on the conveyor must always be within zones of equal length, which can be pictured as lines drawn on a belt conveyor or trays pulled by a chain. For example, in a tilt-tray conveyor, continuously moving trays of fixed size are used to move items. The control system is designed to induct items in such a way that each item is in a separate tray; thus it is a nonaccumulating fixed-window conveyor. In contrast, with random spacing, items can be anywhere on the conveyor section relative to other items. To be inducted, they simply require sufficient space.

Besides these basic types, there are innumerable types of specialized conveyors for special purposes. For example, a specialized indexing conveyor may move forward in increments, always maintaining a fixed distance between the trailing edge of the load ahead and the leading edge of the load behind. Its purpose is to form a "slug" of items, equally spaced apart, to be inducted all together onto a transport conveyor. For the local behavior of some systems—that is, the performance at a particular workstation or induction point—a detailed understanding and accurate model of the physical workings and the control logic are essential for accurate results.

13.2 GOALS AND PERFORMANCE MEASURES

The purpose of simulation is insight, not numbers. Those who purchase and use simulation software and services want to gain insight and understanding into how a new or modified system will work. Will it meet throughput expectations? What happens to response time at peak periods? Is the system resilient to short-term surges? What is the recovery time when short-term surges cause congestion and queueing? What are the staffing requirements? What problems occur? If problems occur, what is their cause and how do they arise? What is the system capacity? What conditions and loads cause a system to reach its capacity?

Simulations are expected to provide numeric measures of performance, such as throughput under a given set of conditions, but the major benefit of simulation comes from the insight and understanding gained regarding system operations. Visualization through animation and graphics provides major assistance in the communication of model assumptions, system operations, and model results. Often, visualization is the major contributor to a model's credibility, which in turn leads to acceptance of the model's numeric outputs. Of course, a proper experimental design that includes the right range of experimental conditions plus a rigorous analysis and, for stochastic simulation models, a proper statistical analysis is of utmost importance for the simulation analyst to draw correct conclusions from simulation outputs.

The major goals of manufacturing-simulation models are to identify problem areas and quantify system performance. Common measures of system performance include the following:

- throughput under average and peak loads;
- system cycle time (how long it takes to produce one part);
- utilization of resources, labor, and machines;
- bottlenecks and choke points;
- queueing at work locations;
- queueing and delays caused by material-handling devices and systems;
- WIP storage needs;
- staffing requirements;
- effectiveness of scheduling systems;
- effectiveness of control systems.

Often, material handling is an important part of a manufacturing system and its performance. Non-manufacturing material-handling systems include warehouses, distribution centers, cross-docking operations, baggage-handling systems at airports and container terminals. The major goals of these non-manufacturing material-handling systems are similar to those identified for manufacturing systems. Some additional considerations are the following:

- how long it takes to process one day of customer orders;
- effect of changes in order profiles (for distribution centers);
- truck/trailer queueing and delays at receiving and shipping docks;
- effectiveness of material-handling systems at peak loads;
- recovery time from short-term surges (for example, with baggage-handling).

13.3 ISSUES IN MANUFACTURING AND MATERIAL-HANDLING SIMULATIONS

There are a number of modeling issues especially important for the achievement of accurate and valid simulation models of manufacturing and material-handling systems. Two of these issues are the proper modeling of downtimes and whether, for some inputs, to use actual system data or a statistical model of those inputs.

13.3.1 Modeling Downtimes and Failures

Unscheduled random downtimes can have a major effect on the performance of manufacturing systems. Many authors have discussed the proper modeling of downtime data (Williams [1994]; Clark [1994]; Law and Kelton [2000]). This section discusses the problems that can arise when downtime is modeled incorrectly and suggests a number of ways to model machine and system downtimes correctly.

Scheduled downtime, such as for preventive maintenance, or periodic downtime, such as for tool replacement, also can have a major effect on system performance. But these downtimes are usually (or should be) predictable and can be scheduled to minimize disruptions. In addition, engineering efforts or new technology might be able to reduce their duration.

There are a number of alternatives for modeling random unscheduled downtime, some better than others:

1. Ignore it.
2. Do not model it explicitly, but increase processing times in appropriate proportion.
3. Use constant values for time to failure and time to repair.
4. Use statistical distributions for time to failure and time to repair.

Of course, alternative (1) generally is not the suggested approach. This is certainly an irresponsible modeling technique if downtimes have an impact on the results, as they do in almost all situations. One situation in which ignoring downtimes could be appropriate, with the full knowledge of the customer, is to leave out those catastrophic downtimes that occur rarely and leave a production line or plant down for a long period of time. In other words, the model would incorporate normal downtimes but ignore those catastrophic downtimes, such as general power failures, snow storms, cyclones, and hurricanes, that occur rarely but stop all production when they do occur. The documented scope of the project should clearly state the assumed operating conditions and those conditions that are not included in the model. If it is generally known that a plant will be closed for some number of snow days per year, then the simulation need not take these downtimes into account, for the effect of any given number of days can easily be factored into the simulation results when making annual projections.

The second possibility, to factor into the model the effect of downtimes by adjusting processing times applied to each job or part, might be an acceptable approximation under limited circumstances. If each job or part is subject to a large number of small delays associated with downtime of equipment or tools, then the total of such delays may be added to the pure processing time to arrive at an adjusted processing time. If total delay time and pure processing time are random in nature, then an appropriate statistical distribution should be used for the total adjusted processing time. If the pure processing time is constant while the total delay time in one cycle is random and variable, it is almost never accurate to adjust the processing time by a constant factor. For example, if processing time is usually 10 minutes but the equipment is subject to downtimes that cause about a 10% loss in capacity, it is not appropriate to merely change the processing time to a constant 11 minutes. Such a deterministic adjustment might provide reasonably accurate estimates of overall system throughput, but will not provide accurate estimates of such local behavior as queue and buffer space needed at peak times. Queueing and short-term congestion are strongly influenced by randomness and variability.

The third possibility, using constant durations for time to failure and time to repair, might be appropriate when, for example, the downtime is actually due to preventive maintenance that is on a fixed schedule. In almost all other circumstances, the fourth possibility, modeling time to failure and time to repair by appropriate statistical distributions, is the appropriate technique. This requires either actual data for choosing a statistical distribution based on the techniques in Chapter 11, or, when data is lacking, a reasonable assumption based on the physical nature of the causes of downtimes.

The nature of time to failure is also important. Are times to failure completely random in nature, a situation due typically to a large number of possible causes of failure? In this case, exponential distribution might provide a good statistical model. Or are times to failure, rather, more regular—typically, due to some major component—say, a tool—wearing out? In this case, a uniform or (truncated) normal distribution could be more nearly appropriate. In the latter case, the mean of the distribution represents the average time to failure, and the distribution places a plus or minus around the mean.

Time to failure can be measured in a number of different ways:

1. by wall-clock time;
2. by machine or equipment busy time;
3. by number of cycle times;
4. by number of items produced.

Breakdowns or failures can be based on clock time, actual usage, or cycles. Note that the word breakdown or failure is used, even though preventive maintenance could be the reason for a downtime. As mentioned, breakdowns or failures can be probabilistic or deterministic in duration.

Actual usage breakdowns are based on the time during which the resource is used. For example, wear on a machine tool occurs only when the machine is in use. Time to failure is measured against machine-busy time and not against wall-clock time. If the time to failure is 90 hours, then the model keeps track of total busy time since the last downtime ended, and, when 90 hours is reached, processing is interrupted and a downtime occurs.

Clock-time breakdowns might be associated with scheduled maintenance—for example, changes of fluids every three months when a complete lubrication is required. Downtimes based on wall-clock time may also be used for equipment that is always busy or equipment that "runs" even when it is not processing parts.

Cycle breakdowns or failures are based on the number of times the resource is used. For example, after every 50 uses of a tool, it needs to be sharpened. Downtimes based on number of cycle times or number of items produced are implemented by generating the number of times or items and, in the model, simply counting until this number is reached. Typical uses of downtimes based on busy time or cycle times may be for maintenance or tool replacement.

Another issue is what happens to a part at a machine when the breakdown or failure occurs. Possibilities include scrapping the part, rework, or simply continuing processing after repair. In some cases—for example, when preventive maintenance is due—the part in the machine may complete processing before the repair (or maintenance activity) begins.

Time to repair can also be modeled in two fundamentally different ways:

1. as a pure time delay (no resources required);
2. as a wait time for a resource (e.g., maintenance person) plus a time delay for actual repair.

Of course, there are many variations on these methods in actual modeling situations. When a repair or maintenance person is a limited resource, the second approach will be a more accurate model and provide more information.

The next example illustrates the importance of using the proper approach for modeling downtimes and of the consequences and inaccurate results that sometimes result from inaccurate assumptions.

Example 13.1: Effect of Downtime on Queueing

Consider a single machine that processes a wide variety of parts that arrive in random mixes at random times. Data analysis has shown that an exponentially distributed processing time with a mean of 7.5 minutes provides a fairly accurate representation. Parts arrive at random, time between arrivals being exponentially distributed with mean 10 minutes. The machine fails at random times. Downtime studies have shown that time-to-failure can be reasonably approximated by an exponential distribution with mean time 1000 minutes. The time to repair the resource is also exponentially distributed, with mean time 50 minutes. When a failure occurs, the current part in the machine is removed from the machine; when the repair has been completed, the part resumes its processing.

When a part arrives, it queues and waits its turn at the machine. It is desired to estimate the size of this queue. An experiment was designed to estimate the average number of parts in the queue. To illustrate the effect of an accurate treatment of downtimes, the model was run under a number of different assumptions. For each case and replication, the simulation run length was 100,000 minutes.

Table 13.1 shows the average number of parts in the queue for six different treatments of the time between breakdowns. For each treatment that involves randomness, five replications of those treatments and the average for those five replications are shown.

Case A ignores the breakdowns. The average number in the queue is 2.31 parts. Across the 5 independent replications, the averages range from 2.05 to 2.70 parts. This treatment of breakdowns is not recommended.

Case B increases the average service time from 7.5 minutes to 8.0 minutes in an attempt to approximate the effect of downtimes. On average, each downtime and repair cycle is 1050 minutes, with the machine down for 50 minutes. Thus the machine is down, on the average in the long run, 50/1050 = 4.8% of total time. Thus, some have argued that downtime has approximately the same effect as increasing the processing

Table 13.1 Average Number of Parts in Queue for Machines with Breakdowns

Case	*1st Rep*	*2nd Rep*	*3rd Rep*	*4th Rep*	*5th Rep*	*Avg Rep*
A. Ignore the breakdowns	2.36	2.05	2.38	2.05	2.70	2.31
B. Increase service time to 8.0	3.32	2.82	3.32	2.81	4.03	3.26
C. All random	4.05	3.77	4.36	3.95	4.43	4.11
D. Random processing, deterministic breakdowns	3.24	2.85	3.28	3.05	3.79	3.24
E. All deterministic						0.52
F. Deterministic processing, Random breakdowns	1.06	1.04	1.10	1.32	1.16	1.13

time of each part by 4.8%, which is about 7.86 minutes. Therefore, an assumed constant 8 minutes per part should be (it is argued) a conservative approach. For this treatment of downtimes, the average number of parts in the queue, over the five replications, is about 3.26 parts. Across the 5 replications, the range is from 2.81 to 4.03 parts. (Note that the variability as shown in the range of values is very small compared to the other cases.) The treatment in Case B might be appropriate under some limited circumstances, but, as was discussed in a previous section, it is not appropriate under the assumptions of this example.

The proper treatment, shown as Case C, treats the randomness in processing and breakdowns properly, with the assumed correct exponential distributions. The average value is about 4.11 parts waiting for the machine. Across the 5 replications, the average queue length ranges from 3.77 to 4.43 parts. The average number waiting differs from that of Case B by almost one part.

Case D is a simplification that treats the processing randomly, but treats the breakdowns as deterministic. The results average about 3.24 parts in the queue. The range of averages is from 2.85 to 3.79 parts, quite a reduction in variability from Case C.

Case E treats all of the times as deterministic. Only one replication is needed, because additional replications (using the same seed) will reproduce the result. The average value in the queue is 0.52 parts, well below the value in Case C, or any other case for that matter. The conclusion: Ignoring randomness is dangerous and leads to totally unrealistic results.

Case F treats arrivals and processing as deterministic, but breakdowns are random. The average number of parts in the queue at the machine is about 1.13. The range is from 1.04 to 1.32 parts. For some machines and processing in manufacturing environments, Case F is the realistic situation: Processing times are constant, and arrivals are regulated—that is, are also constant. The reader is left to consider the inaccuracies that would result from making faulty assumptions regarding the nature of time to failure and time to repair.

In conclusion, there can be significant differences between the estimated average numbers in a queue, based on the treatment of randomness. The results using the correct treatment of randomness can be far different from those using alternatives. Often, one is tempted by the unavailability of detailed data and the availability of averages to want to use average time to failure as if it were a constant. Example 13.1 illustrates the dangers of inappropriate assumptions. Both the appropriate technique to use and the appropriate statistical distribution depend on the available data and on the situation at hand.

As discussed by Williams [1994], the accurate treatment of downtimes is essential for achieving valid models of manufacturing systems. Some of the essential ingredients are the following:

- avoidance of oversimplified and inaccurate assumptions;
- careful collection of downtime data;
- accurate representation of time to failure and time to repair by statistical distributions;
- accurate modeling of system logic when a downtime occurs, in terms both of the repair-time logic and of what happens to the part currently processing.

13.3.2 Trace-driven Models

Consider a model of a distribution center that receives customer orders that must be processed and shipped in one day. One modeling question is how to represent the day's set of orders. A typical order will contain one or more line items, and each line item can have a quantity of one or more pieces. For example, when you buy a new stereo, you might purchase an amplifier, a tuner, and a CD player (all separate line items, each having a quantity of one piece), and 4 identical speakers (another line item with a quantity of 4 pieces). The overall order profile can have a major impact on the performance of a particular system design. A system designed to handle large orders going to a small number of customers might not perform well if order profiles shift toward a larger number of customers (or larger number of separate shipments) with one or two items per order.

One approach is to characterize the order profile by using a discrete statistical distribution for each variable in an order:

1. the number of line items
2. for each line item, the number of pieces.

If these two variables are statistically independent, then this approach might provide a valid model of the order profile. For many applications, however, these two variables may be highly correlated in ways that could be difficult to characterize statistically. For example, an apparel and shoe company has six large customers (the large department stores and discount chains), representing 50% of sales volume, which typically order dozens or hundreds of line items and large quantities of many of the items. At the opposite pole, on any given day approximately 50% of the orders are for one or two pairs of shoes (just-in-time with a vengeance!). For this company, the number of line items in an order is highly positively correlated with the quantity ordered; that is, large orders with a large number of line items also usually have large quantities of many of the line items. And small orders with only a few line items typically order small quantities of each item.

What would happen if the two variables, number of line items and quantity per line item, were modeled by independent statistical distributions? When an order began processing, the model would make two random uncorrelated draws, which could result in order profiles quite different from those found in practice. Such an erroneous assumption could result, for example, in far too large a proportion of orders having one or two line items with large unrealistic quantities.

Another common but more serious error is to assume that there is an average order and to simulate only the number of orders in a day with each being the typical order. In the author's experience, analyses of many order profiles has shown (1) that there is no such thing as a typical order and (2) that there is no such thing as a typical order profile.

An alternative approach, and one that has proven successful in many studies, is for the company to provide the actual orders for a sample of days over the previous year. Usually, it is desirable to simulate peak days. A model driven by actual historical data is called a trace-driven model.

A trace-driven model eliminates all possibility of error due to ignoring or misestimating correlations in the data. One apparent limitation could be a customer's desire, at times, to be able to simulate hypothesized changes to the order profile, such as a higher proportion of smaller orders in terms of both line items and quantities. In practice, this limitation can be removed by adding "dials" to the order-profile portion of the model, so that a simulation analyst can "dial up" more or less of certain characteristics, as desired. One approach is to treat the day's orders as a statistical population from which the model draws samples in a random fashion. This approach makes it easy to change overall order volume without modifying the profile. A second related approach would be to subdivide a day's orders into subgroups based on number of line items, quantities or other numeric parameters, and then sample in a specified proportion from each subgroup. By changing the proportion of each subgroup, different order profiles can be "dialed up" and fed into the model. A third approach is to use factors to adjust the number of daily orders, the number of line items, and/or the quantities. In practice one of these approaches might be as accurate as can be expected for hypothesized future order profiles and might provide a cost effective and reasonably accurate model, especially for testing the robustness of a system design for assumed changes in order characteristics.

Other examples of trace-driven models include the following:

- orders to a custom job shop, using actual historical orders;
- product mix and quantities, and production sequencing, for an assembly line making 100 styles and sizes of hot-water heaters;
- time to failure and downtime, using actual maintenance records;
- Truck arrival times to a warehouse, using gate records.

Whether to make an input variable trace-driven or to characterize it as a statistical distribution depends on a number of issues, including the nature of the variable itself, whether it is correlated with or independent of other variables, the availability of accurate data, and the questions being addressed.

13.4 CASE STUDIES OF THE SIMULATION OF MANUFACTURING AND MATERIAL HANDLING SYSTEMS

The *Winter Simulation Conference Proceedings*, IIE Magazine, Modern Material Handling and other periodicals are excellent sources of information for short cases in the simulation of manufacturing and material-handling systems.

An abstract of some of the papers from past *Winter Simulation Conference Proceedings* will provide some insight into the types of problems that can be addressed by simulation. These abstracts have been paraphrased and shortened where appropriate; our goal is to provide an indication of the breadth of real-world applications of simulation.

Session:	Semiconductor Wafer Manufacturing
Paper:	Modeling and Simulation of Material Handling for Semiconductor Wafer Manufacturing
Authors:	Neal G. Pierce and Richard Stafford
Abstract:	This paper presents the results of a design study to analyze the interbay material-handling systems for semiconductor wafer manufacturing. The authors developed discrete-event simulation models of the performance of conventional cleanroom material handling including manual and automated systems. The components of a conventional cleanroom material-handling system include an overhead monorail system for interbay (bay-to-bay) transport, work-in-process stockers for lot storage, and manual systems for intrabay movement. The authors constructed models and experiments that assisted with analyzing cleanroom material-handling issues such as designing conventional automated material-handling systems and specifying requirements for transport vehicles.

Session:	Simulation in Aerospace Manufacturing
Paper:	Modeling Aircraft Assembly Operations
Authors:	Harold A. Scott
Abstract:	A simulation model is used to aid in the understanding of complex interactions of aircraft assembly operations. Simulation helps to identify the effects of resource constraints on dynamic process capacity and cycle time. To analyze these effects, the model must capture job and crew interactions at the control code level. This paper explores five aspects of developing simulation models to analyze crew operations on aircraft assembly lines:

Representing job precedence relationships
Simulating crew members with different skill and job proficiency levels
Reallocating crew members to assist ongoing jobs
Depicting shifts and overtime
Modeling spatial constraints and crew movements in the production area.

Session:	Control of Manufacturing Systems
Paper:	Discrete Event Simulation for Shop Floor Control
Authors:	J. S. Smith, R. A. Wysk, D. T. Sturrock, S. E. Ramaswamy, G. D. Smith, S. B. Joshi

Abstract: This paper describes an application of simulation to shop floor control of a flexible manufacturing system. The simulation is used not only as an analysis and evaluation tool, but also as a "task generator" for the specification of shop floor control tasks. Using this approach, the effort applied to the development of control logic in the simulation is not duplicated in the development of the control system. Instead the same control logic is used for the control system as was used for the simulation. Additionally, since the simulation implements the control, it provides very high fidelity performance predictions. The paper describes implementation experience in two flexible manufacturing laboratories.

Session: Flexible Manufacturing
Paper: Developing and Analyzing Flexible Cell Systems Using Simulation
Authors: Edward F. Watson and Randall P. Sadowski
Abstract: This paper develops and evaluates flexible cell alternatives to support an agile production environment at a mid-sized manufacturer of industrial equipment. Three work cell alternatives were developed based on traditional flow analysis studies, past experience, and common sense. The simulation model allowed the analyst to evaluate each cell alternative under current conditions as well as anticipated future conditions that included changes to product demand, product mix, and process technology.

Session: Modeling of Production Systems
Paper: Inventory Cost Model for Just-in-Time Production
Authors: Mahesh Mathur
Abstract: This paper presents a simulation model used to compare setup and inventory carrying costs with varying lot sizes. While reduction of lot sizes is a necessary step towards implementation of Just-in-Time (JIT) in a job shop environment, a careful cost study is required to determine the optimum lot size under the present set-up conditions. The simulation model graphically displays the fluctuation of carrying costs and accumulation of set-up costs on a time scale in a dynamic manner. The decision of the optimum lot size can then be based on realistic cost figures.

Session: Analysis of Manufacturing Systems
Paper: Modeling Strain of Manual Work in Manufacturing Systems
Authors: I. Ehrhardt, H. Herper, and H. Gebhardt
Abstract: This paper describes a simulation model that considers manual operations for increasing the effectiveness of planning logistic systems. Even though there is ever increasing automation, there are vital tasks in production and logistics that are still assigned to humans. Present simulation modeling efforts rarely concentrate on the manual activities assigned to humans.

Session: Manufacturing Case Studies
Paper: Simulation Modeling for Quality and Productivity in Steel Cord Manufacturing
Authors: C. H. Turkseven and G. Ertek
Abstract: The paper describes the application of simulation modeling to estimate and improve quality and productivity performance of a steel cord manufacturing system. It focuses on wire fractures, which can be an important source of system disruption.

Session: Manufacturing Analysis and Control
Paper: Shared Resource Capacity Analysis in Biotech Manufacturing

Author: P. V. Saraph

Abstract: This paper discusses an application of simulation in analyzing the capacity needs of a shared resource, the Blast Freezer, at one of the Bayer Corporation's manufacturing facilities. The simulation model was used to analyze the workload patterns, run different workload scenarios, taking into consideration uncertainty and variability, and provide recommendations on a capacity increase plan. This analysis also demonstrated the benefits of certain operational scheduling policies. The analysis outcome was used to determine capital investments for 2002.

Session: Manufacturing Analysis and Control

Paper: Behavior of an Order Release Mechanism in a Make-to-Order Manufacturing System with Selected Order Acceptance

Authors: A. Nandi and P. Rogers

Abstract: The authors used a simulation model to evaluate a controversial policy, namely, holding orders in a pre-shop pool prior to their release to the factory floor. In a make-to-order manufacturing system, if capacity is fixed and exogenous due dates are inflexible, having orders wait in a pre-shop pool may cause the overall due date performance of the system to deteriorate. The model was used to evaluate an alternative approach, the selective rejection of orders for dealing with surges in demand while maintaining acceptable due date performance.

13.5 MANUFACTURING EXAMPLE: AN ASSEMBLY-LINE SIMULATION

This section describes a model of a production line for the final assembly of "gizmos". It then focuses on how simulation can be used to analyze system performance.

13.5.1 System Description and Model Assumptions

At a manufacturing facility, an engineering team has designed a new production line for the final assembly of gizmos. Before making the investment to install the new system, some team members propose using simulation to analyze the system's performance, specifically to predict system throughput (gizmos per 8-hour shift, on the average). In addition, the engineers desire to evaluate potential improvements to the designed system. One such potential improvement is adding buffer space for holding work-in-process (WIP) between adjacent workstations.

The team decides to develop a simulation model and conduct an analysis. The team's primary objective is to predict throughput (completed gizmos per shift on the average) for the given system design and to evaluate whether it meets the desired throughput. In addition, should throughput be less than expected, the team wants to use the model to help in identifying bottlenecks, gaining insight into the system's dynamic behavior and evaluating potential design improvements.

The proposed production line has six workstations and a special rack for WIP storage between adjacent stations. There are four manual stations, each having its own operator, and two automated stations, which share a single operator. The six stations perform production tasks in the following sequence:

Station 1: initial manual station begins final assembly of a new gizmo
Station 2: manual assembly station
Station 3: manual assembly station
Station 4: automatic assembly station
Station 5: automatic testing station
Station 6: manual packing station

At each manual station, an operator loads a gizmo onto a workbench, performs some tasks, and on completion unloads the gizmo and places it into the WIP storage for the next workstation. The operator takes 10 seconds and 5 seconds for the loading and unloading tasks, respectively.

The WIP storage racks between each pair of adjacent stations have limited capacity. If a station completes its tasks on a gizmo but the downstream rack is full, the gizmo must remain in the station, blocking any further work. In the initial design, the WIP storage racks have the capacities shown in Table 13.2. (By assumption, the WIP storage preceding Station 1 is always kept full at 4 units; since it is assumed to always be full, its specific capacity plays no role.) The system design with capacities given in Table 13.2 is called the Baseline configuration.

From time to time, a tool will fail, causing unscheduled downtime or unexpected extra work at a manual or automated station. In addition, all operators are scheduled to take a 30-minute lunch break at the same time. Work is interrupted and resumes where it left off after lunch. This interrupt/resume rule applies to operator tasks including assembly work, parts resupply, and repairs during a downtime.

At the automatic stations, a machine performs an assembly or testing task. The automatic stations might have unscheduled (random) downtimes, but they continue to operate during the operator's lunch break. One operator services both machines to load and unload gizmos (10 seconds and 5 seconds, respectively). After being loaded, a machine processes the gizmo without further operator intervention unless a downtime occurs. At all stations, the operator performs repairs as needed whenever the station experiences a downtime.

Table 13.3 gives the total assembly time and parts resupply times for each station, plus the number of parts in a batch. The assembly time for the manual stations is assumed to vary by plus/minus 2 seconds (uniformly distributed) from the times given in Table 13.3. Parts resupply time does not occur for each gizmo, but rather after a batch of parts has been assembled onto the gizmo. The machines at stations 4 and 5 do not consume parts.

Each station is subject to unscheduled (random) downtime. Manual stations 1–3 have tool failures or other unexpected problems. The automatic stations occasionally jam or have some other problem that requires the assigned operator to fix it. Station 6 (packing) is not subject to these downtimes. Table 13.4

Table 13.2 Capacity of WIP Storage Buffers for Baseline Configuration

Rack Before Station	1	2	3	4	5	6
Buffer Capacity	4	2	2	2	1	2

Table 13.3 Assembly and Parts Resupply Times

Station	*Assembly per Gizmo (Seconds)*	*Part Number*	*Parts Resupply Time (seconds per Batch)*	*No. of Parts per Batch*
1	40	A	10	15
		B	15	10
2	38	C	20	8
		D	15	14
3	38	E	30	25
4	35			
5	35			
6	40	F[a]	30	32

[a]: At station 6, the part number (F) represents the shipping containers.

Table 13.4 Assumptions and Data for Unscheduled Downtimes

Station	*TTF*	*MTTF (Minutes)*	*TTR*	*MTTR (Minutes)*	+/−	*Expected Availability*
1	Exponential	36.0	Uniform	4.0	1.0	90%
2	Exponential	4.5	Uniform	0.5	0.1	90%
3	Exponential	27.0	Uniform	3.0	1.0	90%
4	Exponential	9.0	Uniform	1.0	0.5	90%
5	Exponential	18.0	Uniform	2.0	1.0	90%

shows time to failure (TTF) and time to repair (TTR) distributional assumptions and the assumed mean time to failure (MTTF), mean time to repair (MTTR) and spread (+/−) of repair times. For example, at Station 1, repair time is uniformly distributed with mean 4.0 minutes plus or minus 1.0 minutes—that is, uniformly distributed between 3.0 and 5.0 minutes. Failure can only occur when an operator or machine is working; hence, TTF is modeled by measuring only busy or processing time until a failure occurs.

The primary model output or response is average throughput during the assumed 7.5 working hours per 8-hour shift. The model also measures detailed station utilization, including busy or processing time, idle or starved time (no parts ready for processing), blocked time (part cannot leave station, because downstream WIP buffer is full), unscheduled downtime, and time waiting for an operator.

Station starvation occurs when the operator and station are ready to work on the next gizmo, the just-completed gizmo leaves the station, but upstream conditions cause no gizmo to be ready for this production step. In short, the upstream WIP buffer is empty.

Station blockage occurs when a station completes all tasks on a gizmo, but cannot release the part because the downstream WIP buffer is full. For both starvation and blockage, production time is lost at the given station and cannot be made up.

When an operator services more than one station, as does the operator servicing Stations 4 and 5, it is possible for both stations to need the operator at the same time. This could cause additional delay at the station and is measured by a "wait for operator" state. Blockage, starvation, and wait-for-operator at each station will be measured in order to help explain any throughput shortfall, should it occur, and to assist in identifying potential system improvements.

13.5.2 Presimulation Analysis

A presimulation analysis, taking into account the average station cycle time as well as expected station availability (90%), indicates that each station, if unhindered, can achieve the desired throughput. This initial analysis is carried out as described in this section.

From the assumed downtime data, the team was able to estimate expected station availability, under the (ideal) assumption of no interaction between stations. The expected availability shows each station's individual availability during working (nonlunch, nonbreak) hours, assuming that the operator can always place a completed gizmo into the downstream rack storage and the next gizmo is ready to begin work at the station. Expected availability is computed by MTTF/(MTTF + MTTR), or expected busy time during a downtime "cycle" divided by the length of a downtime cycle (a busy cycle plus a repair cycle) and is given in Table 13.4. This calculation ignores certain aspects of the problem, including the parts resupply times and any delay caused by having only one operator to service both Stations 4 and 5.

The design goal for the modeled system is 390 finished gizmos per 8-hour shift. After taking lunch into account, each shift has up to 7.5 hours of available work time. With unscheduled (random) downtime expected to be 10% of available time, this further reduces working time to 0.90×7.5 hours = 6.75 hours. This implies

Table 13.5 Estimated Total Cycle Time at Each Station

Station	*Formula to Estimate Cycle Time (Seconds)*	*Estimate (Seconds)*
1	10 + 40 + 5 + 10/15 + 15/10	57.2
2	10 + 38 + 5 + 20/8 + 15/14	56.6
3	10 + 38 + 5 + 30/25	54.2
4	10 + 35 + 5	50.0
5	10 + 35 + 5	50.0
6	10 + 40 + 5 + 30/32	55.9

that the station with the slowest total cycle time must be able to produce 390 gizmos in the available 6.75 hours. Therefore the total cycle time per gizmo at each station must not exceed 6.75 hours/390 = 62.3 seconds.

Now, total cycle time consists of assembly, testing or packing time, and parts resupply time (as given in Table 13.3), plus gizmo loading time of 10 seconds and unload time of 5 seconds. Parts resupply is not taken on every gizmo, but rather after a given number of gizmos corresponding to using all parts in a given batch of parts. For example, using the values in Table 13.3 for Station 1, parts resupply will take 10 seconds every 15 gizmos for Part A, plus 15 seconds every 10 gizmos for Part B, for a total time on the average of 10/15 + 15/10 seconds per gizmo.

Using this information, the (minimum) total cycle time for each station is estimated in Table 13.5. These presimulation estimates indicate, first, that each theoretical cycle time is well below the requirement of 62.3 seconds. Secondly, they indicate that Stations 1 and 2 are potential bottlenecks, if there are any.

As the simulation analysis will later show, Station 1 experiences blockage due to Station 2 downtime, and Station 2 occasionally experiences starvation due to downtime at Station 1 and blockage due to downtime at Station 3. These blockage and starvation conditions reduce the available work time below the calculated 90%; hence, for the Baseline Configuration, they reduce the design throughput well below the desired value, 390 gizmos per shift. In summary, a presimulation analysis, although valuable, at best can provide a rough estimate of system performance. As the simulation will show, ignoring blockage and starvation gives an overly optimistic estimate of system throughput.

13.5.3 Simulation Model and Analysis of the Designed System

Using the simulation model, the first experiment was conducted to estimate system performance of the system as designed. The simulation analyst on the team made 10 replications of the model, each having a 2-hour warm-up or initialization followed by a 5-day simulation (each day being 24 hours). A 95% confidence interval was computed for mean throughput per shift:

95% CI for mean throughput: (364.5, 366.8), *or* 365.7 ± 1.14.

With 95% confidence, the model predicts that mean (or long-run average) throughput will be between 364.5 and 366.8 gizmos per 8-hour shift with the system as designed. This is well below the design throughput, 390 gizmos per shift.

The team decided to conduct further analyses to identify possible bottlenecks and potential areas of improvement.

13.5.4 Analysis of Station Utilization

At this point, the team desired to have some explanation of the shortfall in throughput. They suspected that perhaps it had to do with the small WIP buffer capacity and the resulting blockage and starvation. The same

Table 13.6 Detailed Station Utilization for Baseline Configuration

Station	*% Down*	*% Blocked*	*% Starved*	*% Wait for Operator*
1	(8.8,9.6)	(11.4,12.5)	(0.0,0.0)	(0.0,0.0)
2	(8.2,8.4)	(8.0,8.8)	(4.9,5.6)	(0.0,0.0)
3	(7.9,8.6)	(9.9,10.4)	(6.1,6.9)	(0.0,0.0)
4	(8.9,9.6)	(2.0,2.8)	(7.5,8.2)	(13.1,14.4)
5	(8.3,9.0)	(0.0,0.2)	(19.4,20.4)	(3.9,4.7)

model was used to estimate detailed workstation utilization in hopes that it would provide an explanation of throughput shortfall. Table 13.6 contains 95% confidence-interval estimates for the first five workstations for percent of time down, blocked, starved, and waiting for an operator. (Waiting for operator affects only stations 4 and 5, as these two stations share one operator. The other stations have a dedicated operator. In addition to the utilization statistics in Table 13.6, the operators have a 30-minute lunch per 8-hour shift, representing 6.25% of available time.)

From the results in Table 13.6, it appears that blockage and starvation explain some portion of the shortfall in throughput. In addition, another possible explanation surfaces: Station 4 experiences a significant time waiting for the single operator that services stations 4 and 5. This delay at Station 4 could result in a full WIP buffer, which in turn would help explain the blockage at Station 3 preceding it. Percent of time blocked is higher than percent starved for Stations 1 to 3, so it appears that downstream delays could be a significant bottleneck.

The team proposed some possible system improvements:

1. having two operators to service Stations 4 and 5 (instead of the currently proposed one operator);
2. increasing the capacity of some of the WIP buffers;
3. a combination of both.

The expense of additional WIP storage space induced the team to desire to keep total buffer space as small as possible, and to require an additional operator only if absolutely necessary, while achieving the design goal of 390 gizmos per shift.

13.5.5 Analysis of Potential System Improvements

To evaluate the addition of an operator and larger WIP buffers, the model was revised appropriately to allow these changes, and a new analysis was conducted. In this analysis, the capacity of each WIP buffer for Stations 2 – 6 was allowed to increase by one unit above the Baseline value given in Table 13.2. In addition, the effect of a second operator at Stations 4 and 5 is considered. These possibilities result in a total of 64 scenarios or model configurations. (Why?) Making 10 replications per scenario results in a total of 640 simulation runs.

To facilitate the analysis, the team decided to use the Common Random Number technique discussed in Section 12.1.3. To implement it with proper synchronization, each source of random variability was identified and assigned a dedicated random-number stream. In this model, processing time, TTF, and TTR are modeled by statistical distributions at each of the six workstations. Therefore, a total of 18 random-number streams were defined, with 3 used at each workstation. In this way, in each set of runs, each workstation experienced the same workload and random downtimes no matter which configuration was being simulated. For a given number of replications the CRN technique, also known as correlated sampling, is expected to give shorter confidence intervals for differences in system performance.

The model configurations with the most improvement in system throughput, compared with the Baseline configuration, are shown in Table 13.7. These configurations were chosen for further evaluation because

Table 13.7 Improvement in System Throughput for Alternative Configurations

Number of Operators Stations 4 & 5	*Buffer Capacities*						*Increase in Mean Throughput per Shift (Compared to Baseline)*		
	Buffer 2	*Buffer 3*	*Buffer 4*	*Buffer 5*	*Buffer 6*	*Total*	*Ave. Diff.*	*CI Low*	*CI High*
2	3	3	3	2	2	13	31.7	30.3	33.1
2	3	3	3	2	3	14	31.7	30.4	33.0
2	3	3	2	2	3	13	30.0	28.6	31.3
2	3	3	3	1	3	13	29.8	28.6	31.0
2	3	3	2	2	2	12	29.7	28.1	31.3
2	3	3	3	1	2	12	29.5	28.1	31.0
2	3	3	2	1	3	12	26.6	25.4	27.9
2	2	3	3	2	2	12	26.6	25.1	28.1
2	2	3	3	2	3	13	26.6	25.0	28.1
2	3	2	3	2	3	13	26.5	25.0	28.0
2	3	2	3	2	2	12	26.4	25.3	27.5
2	3	3	2	1	2	11	26.3	25.1	27.5

each shows a potential improvement in throughput of approximately 25 units or more—that is, the lower end of the 95% confidence interval is 25 or higher. The values shown for "Ave Diff" represent the increase in throughput compared to the Baseline configuration. Recall that the Baseline throughput was previously estimated, with 95% confidence, to be in the interval (364.5, 366.8). Being conservative, the engineering team would like to see an improvement of 390 − 364.5 = 25.5 gizmos per shift. The top six configurations in Table 13.7 have a lower confidence interval larger than 25.5 and hence are likely candidates for achieving the desired throughput. Interpreted statistically: The lower end of the confidence interval is larger than 25.5, so the results yield a 95% confidence that mean throughput will increase by 25.5 or more in the top six configurations listed in Table 13.7.

Note that all the most improved configurations include two operators at Stations 4 and 5. The simulation results for configurations with one operator (not shown here) indicate that a 390 throughput cannot be achieved with one operator, at least not with the buffer sizes considered.

Some configurations can be ruled out because a less expensive option achieves a similar throughput. Consider, for example, the first two configurations in Table 13.7. They are identical except for Buffer 6 capacity. Since WIP buffer capacity is expensive, the smaller total buffer capacity will be the less expensive option. Clearly, there is no need to expand from 2 to 3 units at Buffer 6. The "Total" column can assist in quickly ruling out configurations that do no better than a similar one with smaller total buffer capacity.

The model configuration that increases throughput by 25.5 or better and has the smallest total buffer capacity is the fifth one in Table 13.7, with capacities of (3,3,2,2,2) for Buffers 2 to 6, respectively. On these considerations, this system design becomes the team's top candidate for further evaluation. The next step (not included here) would be to conduct a financial analysis of each alternative configuration.

13.5.6 Concluding Words: The Gizmo Assembly-Line Simulation

Real-life examples similar to this example model include assembly lines for automotive parts and automobile bodies, automotive pollution-control assemblies, consumer items such as washing machines, ranges, and dishwashers, and any number of other assembly operations with a straight flow and limited buffer space between workstations. Similar models and analyses may also apply to a job shop with multiple products, variable routing, and limited work-in-process storage.

13.6 SUMMARY

This chapter introduced some of the ideas and concepts most relevant to manufacturing and material handling simulation. Some of the key points are the importance of modeling downtimes accurately, the advantages of trace-driven simulations with respect to some of the inputs, and the need in some models for accurate modeling of material-handling equipment and the control software.

REFERENCES

BANKS, J. [1994], "Software for Simulation," in *1994 Winter Simulation Conference Proceedings*, eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Association for Computing Machinery, New York, pp. 26–33.

CLARK, G. M. [1994], "Introduction to Manufacturing Applications," in *1994 Winter Simulation Conference Proceedings*, eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Association for Computing Machinery, New York, pp. 15–21.

EHRHARDT, I., H. HERPER, AND H. GEBHARDT [1994], "Modelling Strain of Manual Work in Manufacturing Systems," in *1994 Winter Simulation Conference Proceedings*, eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Association for Computing Machinery, New York, pp. 1044–1049.

LAW, A. M. AND W. D. KELTON [2000], *Simulation Modeling and Analysis*, 3d ed., McGraw–Hill, New York.

NANDI, A., AND P. ROGERS [2003], "Behavior of an Order Release Mechanism in a Make-to-Order Manufacturing System with Selected Order Acceptance," in *2003 Winter Simulation Conference Proceedings*, eds. S. E. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, Association for Computing Machinery, New York, pp. 1251–1259.

PIERCE, N. G., AND R. STAFFORD [1994], "Modeling and Simulation of Material Handling for Semiconductor Wafer Fabrication," in *1994 Winter Simulation Conference Proceedings*, eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Association for Computing Machinery, New York, pp. 900–906.

SARAPH, P. V. [2003], "Shared Resource Capacity Analysis in Biotech Manufacturing," in *2003 Winter Simulation Conference Proceedings*, eds. S. E. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, Association for Computing Machinery, New York, pp. 1247–1250.

SCOTT, H. A. [1994], "Modeling Aircraft Assembly Operations," in *1994 Winter Simulation Conference Proceedings*, eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Association for Computing Machinery, New York, pp. 920–927.

SMITH, J. S., *et al.* [1994], "Discrete Event Simulation for Shop Floor Control," in *1994 Winter Simulation Conference Proceedings*, eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Association for Computing Machinery, New York, pp. 962–969.

TURKSEVEN, C. H., AND G. ERTEK [2003], "Simulation Modeling for Quality and Productivity in Steel Cord Manufacturing," in *2003 Winter Simulation Conference Proceedings*, eds. S. E. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, Association for Computing Machinery, New York, pp. 1225–1229.

WATSON, E. F., AND R. P. SADOWSKI [1994], "Developing and Analyzing Flexible Cell Systems Using Simulation," in *1994 Winter Simulation Conference Proceedings*, eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Association for Computing Machinery, New York, pp. 978–985.

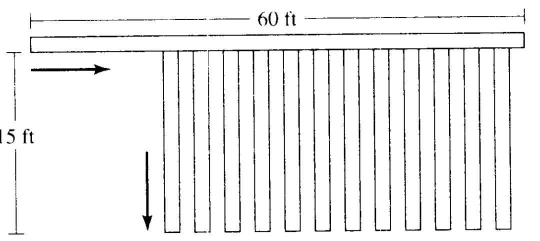
WILLIAMS, E. J. [1994], "Downtime Data—Its Collection, Analysis, and Importance," in *1994 Winter Simulation Conference Proceedings*, eds. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Association for Computing Machinery, New York, pp. 1040–1043.

EXERCISES

Instructions to the student: Many of the following exercises contain material-handling equipment such as conveyors and vehicles. The student is expected to use any simulation language or simulator that supports modeling conveyors and vehicles at a high level.

Some of the following exercises use the uniform, exponential, normal, or triangular distributions. Virtually all simulation languages and simulators support these, plus other distributions. The use of the first three distributions was explained in the note to the exercises in Chapter 4; the use of the triangular is explained in the exercise that requires it. For reference, the properties of these distributions, plus others used in simulation, are given in Chapter 5, and random-variate generation is covered in Chapter 8.

1. A case sortation system consists of one infeed conveyor and 12 sortation lanes, as shown in the following schematic (not to scale):



Cases enter the system from the left at a rate of 50 per minute at random times. All cases are 18 by 12 inches and travel along the 18 inch dimension. The incoming mainline conveyor is 20 inches wide and 60 feet in length (as shown). The sortation lanes are numbered 1 to 12 from left to right, and are 18 inches wide and 15 feet in length, with 2 feet of spacing between adjacent lanes. (Estimate any other dimensions that are needed.) The infeed conveyor runs at 180 feet/minute, the sortation lanes at 90 feet/minute. All conveyor sections are accumulating, but, upon entrance at the left, incoming cases are at least 2 feet apart from leading edge to leading edge. On the sortation lanes, the cases accumulate with no gap between them.

Incoming cases are distributed to the 12 lanes in the following proportions:

1	6%	7	11%
2	6%	8	6%
3	5%	9	5%
4	24%	10	5%
5	15%	11	3%
6	14%	12	0%

The 12th lane is an overflow lane; it is used only if one of the other lanes fill and a divert is not possible.

At the end of the sortation lanes, there is a group of operators who scan each case with a bar-code scanner, apply a label and then place it on a pallet. Operators move from lane to lane as necessary to avoid allowing a lane to fill. There is one pallet per lane, each holding 40 cases. When a pallet is full, assume a new empty one is immediately available. If a lane fills to 10 cases and another case arrives at the divert point, this last case continues to move down the 60-foot mainline conveyor and is diverted into lane 12, the overflow lane.

Assume that one operator can handle 8.5 cases per minute, on the average. Ignore walking time and assignment of an operator to a particular lane; in other words, assume the operators work as a group uniformly spread over all 12 lanes.

(a) Set up an experiment that varies the number of operators and addresses the question: How many operators are needed? The objective is to have the minimum number of operators but also to avoid overflow.

(b) For each experiment in part (a), report the following output statistics:

Operator utilization
Total number of cases palletized
Number of cases palletized by lane
Number of cases to the overflow lane

(c) For each experiment in part a, verify that all cases are being palletized. In other words, verify that the system can handle 50 cases per minute, or explain why it cannot.

2. Redo Exercise 1 to a greater level of detail by modeling operator walking time and operator assignment to lanes. Assume that operators walk at 200 feet per minute and that the walking distance from one lane to the next is 5 feet. Handling time per case is now assumed to be 7.5 cases per minute. Devise a set of rules that can be used by operators for lane changing. (For example, change lanes to that lane with the greatest number of cases only when the current lane is empty or the other lane reaches a certain level.) Assume that each operator is assigned to a certain number of adjacent lanes and handles only those lanes. However, if necessary, two operators (but no more) may be assigned to one lane—that is, operator assignments may overlap.

(a) If your lane-changing rule has any numeric parameters, experiment to find the best settings. Under these circumstances, how many operators are needed? What is the average operator utilization?

(b) Does a model that has more detail, as does Exercise 2a when compared to Exercise 1, always have greater accuracy? How about this particular model? Compare the results of Exercise 2a to the results for Exercise 1. Are the same or different conclusions drawn?

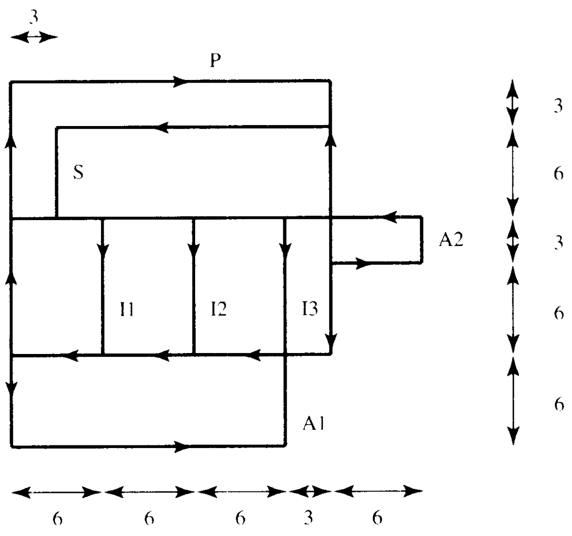
(c) Devise a second lane-changing rule. Compare results between the two rules. Compare total walking time or percent of time spent walking between the two rules.
Suggestion: A lane-changing rule could have one or two "triggers". A one-trigger rule might state that, if a lane reached a certain level, the operator moved to that lane. (Without modification, such a rule could lead to excessive operator movement, if two lanes had about the same number of cases near the trigger level.) A two-trigger rule might state that, if a lane reached a certain level and the operator's current lane became empty, then change to the new lane; but if a lane reaches a specified higher "critical" level, then the operator immediately changes lanes.

(d) Compare your results with those of other students who may have used a different lane-changing rule.

3. Parts carried by the AGV system arrive through three intersections are

Intersection	*Interarrival Time (Minutes)*
I1	10 ± 4
I2	8 ± 2
I3	20 ± 6

The parts are to be assembled in any one of the assembly stations A1 or A2. The assembly time is



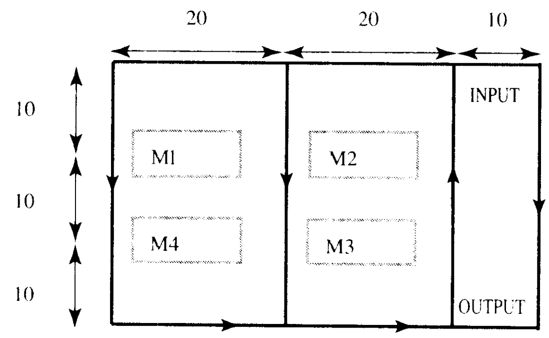
7 ± 2 minutes. After assembly, parts are sent to the output station P. If both A1 and A2 are free, parts have an equal probability of going to either A1 or A2. AGV is required to take the arriving part to assembly station and assembled part to output station. Once AGV becomes free, it responds to any waiting call, otherwise it is sent to staging area S. All links are unidirectional and the distances are shown in meters. The AGV speed is 40 meters per minute. Delay the start of the assembly operations for 30 minutes after parts start arriving to allow a buildup of parts. Simulate the system for 10,000 minutes. Determine the number of AGVs required to ensure that there is always a part available for the assembly operations.

4. Redo the simulation with the assumption that the assembly times are different in A1 and A2 as

Assembly Station	*Assembly Time (Minutes)*
A1	9 ± 2
A2	7 ± 2

Hence if both A1 and A2 are free, the part is taken to the assembly station A2.

5. In a machine shop, there are four machines M1, M2, M3, and M4. They are identical in all respects and served by AGVs. Parts arrive with interarrival time following exponential with a mean of 5 minutes. Machines do not have any buffer space. So an arriving part at the input area must first gain access to a free machine before it can be moved to the machine. When a machine finishes an operation, an AGV is requested and the machine is to be made free only after the part has been picked up by the AGV. Processing time follows normal with a mean of 8 minutes and a standard deviation of 2 minutes. It takes 30 seconds to load and unload the parts. AGV takes the finished parts to the output station and the AGV is free to respond to other requests, or is sent to the input area that serves as a staging area. The AGVs move at a speed of 25 meters per minute. The dimensions shown are in meters, and the intersections are 0 meter in length. Simulate this system for 2,500 minutes. Change the number of AGVs and analyze the impact on parts waiting time.



6. Reconsider Exercise 5. Assume that two types of parts are arriving and the parts are to be processed in more than one machine. Parts arrive with interarrival time following exponential with a mean of 5 minutes. The sequence of operation and the percentage of part types are

Part Type	*Percentage*	*Sequence*
A	60	M1, M2, M4
B	40	M2, M3

Process time at the machines are

Machine	*Process Time*
M1	$N(8,2)$
M2	4 ± 2
M3	$N(8,1)$
M4	9 ± 2

Simulate this system for 2,500 minutes. Change the number of AGVs and analyze the impact on parts waiting time.

7. Develop a model for Example 13.1 and attempt to reproduce qualitatively the results found in the text regarding different assumptions for simulating downtimes. Do not attempt to get exactly the same numerical results, but rather to show the same qualitative results.

   (a) Do your models support the conclusions discussed in the text? Provide a discussion and conclusions.

   (b) Make a plot of the number of entities in the queue versus time. Can you tell when failures occurred? After a repair, about how long does it take for the queue to get back to "normal"?

8. In Example 13.1, the failures occurred at low frequency compared with the processing time of an entity. Time to failure was 1000 minutes, and interarrival time was 10 minutes, implying that few entities would experience a failure. But, when an entity did experience a failure (of 50 minutes, on average), it was several times larger than the processing time of 7.5 minutes.

   Redo the model for Example 13.1, assuming high-frequency failures. Specifically, assume that the time to failure is exponentially distributed, with mean 2 minutes, and the time to repair is exponentially distributed, with mean 0.1 minute or 6 seconds. As compared with the low-frequency case, entities will tend to experience a number of short downtimes.

For low-frequency versus high-frequency downtimes, compare the average number of downtimes experienced per entity, the average duration of downtime experienced, the average time to complete service (including downtime, if any), and the percent of time down.

Note that the percentage of time the machine is down for repair should be the same in both cases:

$$50/(1000 + 50) = 4.76\%$$
$$6 \text{ sec}/(2\text{min}+6 \text{ sec}) = 4.76\%$$

Verify percentage downtime from the simulation results. Are the results identical? ... close? Should they be identical, or just close? As the simulation run-length increases, what should happen to percentage of time down?

With high-frequency failures, do you come to the same conclusions as were drawn in the text regarding the different ways to simulate downtimes? Make recommendations regarding how to model low-frequency versus high-frequency failures.

9. Redo Exercise 11 (based on Example 13.1), but with one change: When an entity experiences a downtime, it must be reprocessed from the beginning. If service time is random, take a new draw from the assumed distribution. If service time is constant, it starts over again. How does this assumption affect the results?

10. Redo Exercise 11 (based on Example 13.1), but with one change: When an entity experiences a downtime, it is scrapped. How does scrapping entities on failure affect the results in the low-frequency and in the high-frequency situations? What are your recommendations regarding the handling of low-versus high-frequency downtimes when parts are scrapped?

11. Sheets of metal pass sequentially through 4 presses: shear, punch, form, and bend. Each machine is subject to downtime and die change. The parameters for each machine are as follows:

Press	*Process Rate (per min.)*	*Time to Failure (min.)*	*Time to Repair (min.)*	*No. of Sheets to a Die Change (no. sheets)*	*Time to Change Die (min.)*
Shear	4.5	100	8	500	25
Punch	5.5	90	10	400	25
Form	3.8	180	9	750	25
Bend	3.2	240	20	600	25

Note that processing time is given as a rate—for example, the shear press works at a rate of 4.5 sheets per minute. Assume that processing time is constant. The automated equipment makes the time to change a die fairly constant, so it is assumed to be always 25 minutes. Die changes occur between stamping of two sheets after the number shown in the table have gone through a machine. Time to failure is assumed to be exponentially distributed, with the mean given in the table. Time to repair is assumed to be uniformly distributed, with the mean taken from the table and a half-width of 5 minutes. When a failure occurs, 20% of the sheets are scrapped. The remaining 80% are reprocessed at the failed machine after the repair.

Assume that an unlimited supply of material is available in front of the shear press, which processes one sheet after the next as long as there is space available between itself and the next machine, the punch press.

In general, one machine processes one sheet after another continuously, stopping only for a downtime, for a die change, or because the available buffer space between itself and the next machine becomes full. Assume that sheets are taken away after bending at the bend press. Buffer space is divided into 3 separate areas, one between the shear and the punch presses, the second between the punch and form presses, and the last between form and bend.

(a) Assume that there is an unlimited amount of space between machines. Run the simulation for 480 hours (about 1 month with 24 hour days, 5 days per week). Where do backups occur? If the total buffer space for all three buffers is limited to 15 sheets (not counting before shear or after bend), how would you recommend dividing this space among the three adjacent pairs of machines? Does this simulation provide enough information to make a reasonable decision?

(b) Modify the model so that there is a finite buffer between adjacent machines. When the buffer becomes full and the machine feeding the buffer completes a sheet, the sheet is not able to exit the machine. It remains in the machine blocking additional work. Assume that total buffer space is 15 sheets for the 3 buffers.

Use the recommendation from part (a) as a starting point for each buffer size. Attempt to minimize the number of runs. You are allowed to experiment with a maximum of 3 buffer sizes for each buffer. (How many runs does this make?) Run a set of experiments to determine the allocation of buffer space that maximizes production. Simulate each alternative for at least 1000 hours.

Report total production per hour on the average, press utilization (broken down by percentage of time busy, down, changing dies, and idle), and average number of sheets in each buffer.

14

Simulation of Computer Systems

It is only natural that simulation is used extensively to simulate computer systems, because of their great importance to the everyday operations of business, industry, government, and universities. In this chapter, we look at the motivations for simulating computer systems, the different types of approaches used, and the interplay between characteristics of the model and implementation strategies. We begin the discussion by looking at general characteristics of computer-system simulations. Next, we lay the groundwork for investigating simulation of computer systems by looking at various types of simulation tools used to perform those simulations. In section 14.3, we describe different ways that input is presented or generated for these simulations. We next work through an example of a high-level computer system one might simulate, paying attention to problems of model construction and output analysis. In section 14.5, we turn to the central processing unit (CPU) and point out what is generally simulated and how. Following this, we consider simulation of memory systems, in section 14.6.

14.1 INTRODUCTION

Computer systems are incredibly complex. A computer system exhibits complicated behavior at time scales from the time to "flip" a transistor's state (on the order of 10^{-11} seconds) to the time it takes a human to interact with it (on the order of seconds or minutes). Computer systems are designed hierarchically, in an effort to manage this complexity. Figure 14.1 illustrates the point. At a high level of abstraction (the system level), one might view computational activity in terms of tasks circulating among servers, queueing for service when a server is busy. A lower level in the hierarchy can view the activity as being among components of a given processor (its registers, its memory hierarchy). At a lower level still, one views the activity of functional units that together make up a central processing unit, and, at an even lower level, one can view the logical circuitry that makes it all happen.

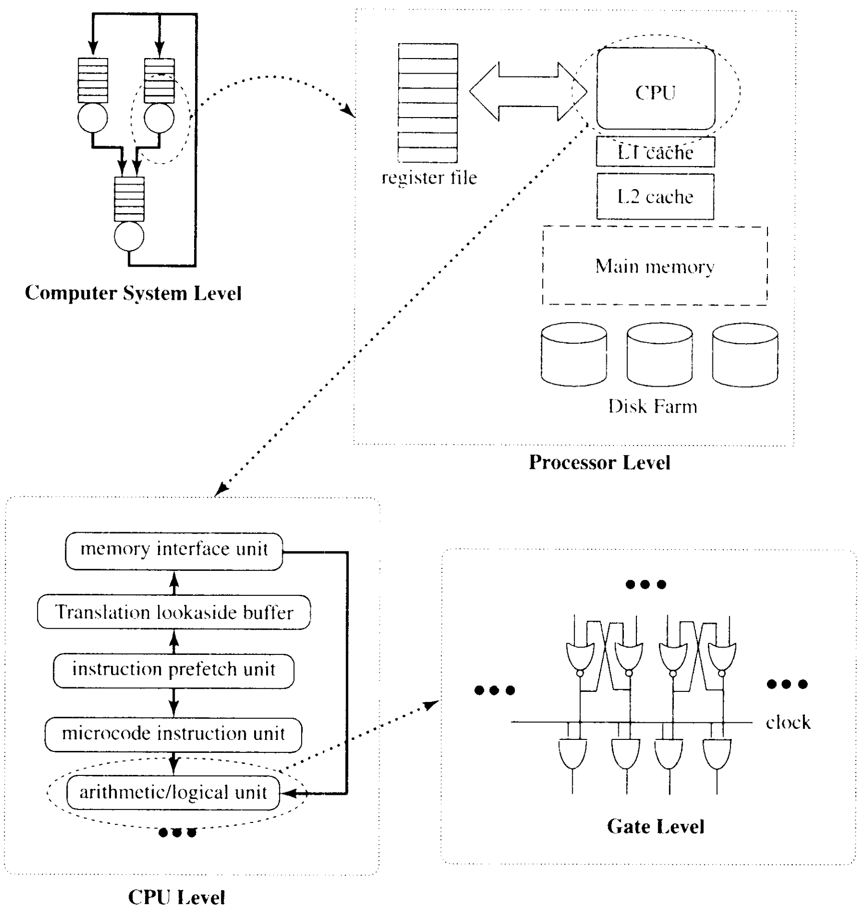


Figure 14.1 Different levels of abstraction in computer systems.

Simulation is used extensively at every level of this hierarchy, with some results from one level being used at another. For instance, engineers working on designing a new chip will begin by partitioning the chip functionally (e.g., the subsystem that does arithmetic, the subsystem that interacts with memory, and so on), establish interfaces between the subsystems, then design and test the subsystems individually. Given a subsystem design, the electrical properties of the circuit are first studied by using a circuit simulator that solves differential equations describing electrical behavior. At this level, engineers work to ensure the correctness of signals' timing throughout the circuit and to ensure that the electrical properties fall within the parameters intended by the design. Once this level of validation has been achieved, the electrical behavior is abstracted into logical behavior (e.g., signals formerly thought of as electrical waveforms are now thought of as logical 1's and 0's). A different type of simulator is next used to test the correctness of the circuit's logical behavior. A common testing technique is to present the design with many different sets of logical inputs ("test vectors") for which the desired logical outputs are known. Discrete-event simulation is used to evaluate the logical response of the circuit to each test vector and is also used to evaluate timing (e.g., the time required to load a register with a datum from the main memory). Once a chip's subsystems are designed and tested, the designs are integrated, and then the whole system is subjected to testing, again by simulation.

At a higher level, one simulates by using functional abstractions. For instance, a memory chip could be modeled simply as an array of numbers, and a reference to memory as just an indexing operation. A special type of description language exists for this level, called "register-transfer-language" (see, for instance, Mano 1993). This is like a programming language, with reassigned names for registers and other hardware specific entities and with assignment statements used to indicate data transfer between hardware entities. For example, the following sequence loads into register r3 the data whose memory address is in register r6, subtracts one from it, and writes the result into the memory location that is word adjacent (a word in this example is 4 bytes in size) to the location first read:

```
r3 = M[r6];
r3 = r3-1;
r6 = r6+4;
M[r6] = r3;
```

A simulator of such a language might ascribe deterministic time constants to the execution of each of these statements. This is a useful level of abstraction to use when one needs to express sequencing of data transfers at a low level, but not so low as the gates themselves. The abstraction makes sense when one is content to assume that the memory works and that the time to put a datum in or out is a known constant. The "known constant" is a value resulting from analysis at a lower level of abstraction. Functional abstraction is also commonly used to simulate subsystems of a central processing unit (CPU), in the study of how an executing program exercises special architectural features of the CPU.

At a higher level still, one might study how an Input–Output (I/O) system behaves in response to execution of a computer program. The program's behavior may be abstracted to the point of being *modeled*, but with some detailed description of I/O demands (e.g., with a Markov chain that with some specificity describes an I/O operation as the Markov chain transitions). The behavior of the I/O devices may be abstracted to the point that all that is considered is how long it takes to complete a specified I/O operation. Because of these abstractions, one can simulate larger systems, and simulate them more quickly. Continuing in this vein, at a higher level of abstraction still, one dispenses with specificity altogether. The execution of a program is modeled with a randomly sampled CPU service interval; its I/O demand is modeled as a randomly sampled service time on a randomly sampled I/O device.

Different levels of abstraction serve to answer different sorts of questions about a computer system, and different simulation tools exist for each level. Highly abstract models rely on stochastically modeled behavior to estimate high-level system performance, such as throughput (average number of "jobs" processed per unit time) and mean response time (per job). Such models can also incorporate system failure and repair and can estimate metrics such as mean time to failure and availability. Less abstract models are used to evaluate specific systems components. A study of an advanced CPU design might be aimed at estimating the throughput (instructions executed per unit time); a study of a hierarchical memory system might seek to estimate the fraction of time that a sought memory reference was found immediately in the examined memory. As we have already seen, more detailed models are used to evaluate functional correctness of circuit design.

14.2 SIMULATION TOOLS

Hand in hand with different abstraction levels, one finds different tools used to perform and evaluate simulations. We next examine different types of tools and identify important characteristics about their function and their use.

An important characteristic of a tool is how it supports model building. In many tools, one constructs networks of components whose local behavior is already known and already programmed into the tool. This is a powerful paradigm for complex model construction. At the low end of the abstraction hierarchy, electrical circuit simulators and gate-level simulators are driven by network descriptions. Likewise, at the high end of the abstraction hierarchy, tools that simulate queueing networks and Petri nets are driven by network descriptions, as are sophisticated commercial communication-system simulators that have extensive libraries of preprogrammed protocol behaviors. Some of these tools allow one to incorporate user-programmed behavior, but it appears this is not the norm as a usage pattern.

A very significant player in computer-systems design at lower levels of abstraction is the VHDL language (e.g., see Ashenden [2001]). VHDL is the result of a U.S. effort in the 1980's to standardize the languages used to build electronic systems for the government. It has since undergone the IEEE standardization process and is widely used throughout the industry. As a language for describing digital electronic systems, VHDL serves both as a design specification and as a simulation specification. VHDL is a rich language, full both of constructs specific to digital systems and the constructs one expects to find in a procedural programming language. It achieves its dual role by imposing a clear separation between system topology and system behavior. Design specification is a matter of topology; simulation specification is a matter of behavior. Libraries of predefined subsystems and behaviors are widely available, but the language itself very much promotes user-defined programmed behavior. VHDL is also innovative in its use of abstract interfaces (e.g., to a functional unit) to which different "architectures" at different levels of abstraction may be attached. For instance, the interface to the Arithmetic Logical Unit (ALU) would be VHDL "signals" that identify the input operands, the operation to be applied to them, and the output. One could attach to this interface an architecture that in a few lines of code just performs the operation—if an addition is specified, just one VHDL statement assigns the output signal to be the sum (using the VHDL addition operator) of the two input signals. An alternative architecture could completely specify the gate-level logical design of the ALU. Models that interact with the ALU interface cannot tell how the semantics of the interface are implemented. This separation of interface from architecture supports modular construction of models and allows one to validate a new submodel architecture by comparing the results it returns to the interface with those returned by a different architecture given the same inputs. A substantive treatment of VHDL is well beyond the scope of this book. VHDL is widely used in the electrical and computer engineering community, but is hardly used outside of it.

One drawback to VHDL is that it is a big language, requires a substantial VHDL compiler, and vendors typically target the commercial market at prices that exclude academic research. Of course, other simulation languages exist, and this text describes several in Chapter 4. Such languages are good for modeling certain types of computer systems at a high level, but are not designed or suited for expression of computer-systems modeling at lower levels of the abstraction hierarchy. As a result, when computer scientists need to simulate specialized model behavior, they will often write a simulation (or a simulator) from scratch. For example, if a new policy for moving data between memories in a hierarchy is to be considered, an existing language will not have that policy preprogrammed; when a new architectural feature in a CPU is designed, the modeler will have to describe that feature and its interaction with the rest of the CPU, using a general programming language. A class of tools exists that use a general programming language to express simulation-model behavior, among them SimPack (Fishwick [1992]), C++SIM (Little and McCue [1994]), CSIM (Schwetman [1986]), Awesime (Grunwald [1995]), and SSF (Cowie *et al.* [1999]). This type of tool defines objects and libraries for use with such languages as C, C++, Java. Model behavior is expressed as a computer program that manipulates these predefined objects. The technique is especially powerful when used with object-oriented languages, because the tool can define base-class objects whose behavior is extended by the modeler.

Some commercial simulation languages do support interaction with general programming languages; however, simulation languages are not frequently used in the academic computer-science world. Cost is a partial explanation. Commercial packages are developed with commercial needs and commercial budgets in mind, yet computer scientists can usually develop what they need relatively quickly, themselves. Another explanation is a matter of emphasis: Simulation languages tend to include a rich number of predefined simulation objects and actions and allow access to a programming language to express object behavior; a simulation model is expressed primarily in the constructs of the simulation language, and the model is evaluated either by compiling the model (using a simulation-language-specific compiler) and running it or by using a simulation-language-specific interpreter.

One of the many advantages to such an approach is that the relative rigidity of the programming model makes possible graphical model building, thereby raising the whole model-building endeavor to a higher level of abstraction. Some tools have so much preprogrammed functionality that it is possible to design and run a model without writing a single line of computer code.

By contrast, programming languages with simulation constructs tend to define a few elemental simulation objects; a simulation model is expressed principally via the notions and control flow of the general programming language, with references to simulation objects interspersed. To evaluate the model, one compiles or interprets the program, using a compiler or interpreter associated with the general programming language, as opposed to one associated with the simulation language. The former approach supports more rapid model development in contexts where the language is tuned to the application; the latter approach supports much greater generality in the sorts of models that can be expressed.

Among tools supporting user-programmed behavior, a fundamental characteristic is the worldview that is supported. In the following two subsections, we look closely at process orientation as it is expressed in SSF, then at an event-oriented approach using a Java base framework.

14.2.1 Process Orientation

A process-oriented view (see Chapter 3) implies that the tool must support separately schedulable threads of control. Threading is a fundamental concept in programming, and a discussion of its capabilities and implementation serves to highlight important issues in simulation modeling. Fundamentally, a "thread" is a separately schedulable unit of execution control, implemented as part of a single executing process (as seen by the operating system; see Nutt [2004]). An operating system has the notion of separate processes (which might interact), which typically have their own separate and independent memory spaces. A group of threads operate in the same process memory space, with each thread having allocated to it a relatively small portion of that space for its own use. That space is used to contain the thread's *state*, which is the full set of all information needed to restart the thread after it is suspended. State would include register values and the thread's runtime stack, which holds variables that are local to the procedures called by the thread. Once a thread is given control, it runs until it yields up control, either via an explicit statement that serves simply to relinquish control or by blocking until signaled by another thread to continue.

These ideas are made more concrete by discussing them in the context of a Java implementation of SSF. Java defines the `Thread` class; a subclass of `Thread` defines the `execute` method, which is defined in the thread body. Threads coordinate with each other through "locks," which provide mutually exclusive access to code segments. Every instance of a Java object has an associated lock (and almost every variable in Java is an object). A thread tries to execute a code fragment protected by the lock for object `obj` via the Java statement

```
synchronized(obj) { /* code fragment */ }
```

A thread must acquire the lock before executing the code fragment, and only one thread has the lock at a time. A thread that executes a `synchronized` statement at an instant at which another thread holds the lock blocks—which could mean suspension, depending on the thread scheduler. Java threads can also coordinate through `wait` and `notify` method calls, also associated with an object's lock. A thread that executes `obj.wait()` suspends. Actually, multiple threads can execute `obj.wait()`, and each will suspend. Eventually some thread executes `obj.notify()`, and the thread scheduler releases one of the suspended threads to continue.

These notions can be used to implement process orientation in a Java simulator. Each simulation process derives from the Java Thread class. One additional thread will maintain an event list; processing for that thread involves removing the least-time event from the event list, reanimating the simulation process thread (or threads) associated with that event, and blocking until those threads have completed. While a process thread is executing, it may cause additional events to be inserted into the scheduler thread's event list. When a process thread completes, it needs to block and to signal the scheduler thread that it is finished. We accomplish all this by using two locks per simulation process. One of these locks is the one Java provides automatically for every object (and a simulation process thread is an object). The other lock is a variable each simulation object defines, which we'll call `lock`. A suspended process thread blocks on a call `lock.wait();` it remains blocked there until the scheduling thread executes `notify()` on that same object variable. After the scheduler does this, it blocks by calling `wait()` on the simulation process object's own built-in lock. So the simulation process thread notifies the scheduler that it is finished by calling `notify()` on its own built-in lock.

SSF code we discussed earlier in Chapter 4 (Figures 4.14 and 4.15) illustrates some of these points. Recall that this code models a single server with exponentially distributed interarrival times and positive normal service times. A cursory glance shows the model to be legitimate Java code that uses SSF base classes.

SSF defines five base classes around which simulation frameworks are built (discussed in Chapter 4). The key one for discussing process orientation is the **process** class; derived classes **Arrivals** in Figure 4.14 and **Server** in Figure 4.15 are examples of it. The base class specifies that method **action** be the thread body; each derived class overrides the base-class definition to specify its own thread's behavior. Every object of a given class derived from **process** defines a separate thread of control, but all execute the same thread code body.

The `waitFor` statement used in `Arrival`'s thread body suspends the thread; its argument specifies how long in simulation time the thread suspends. The Java thread-based scheduling mechanism we described earlier enables implementation of `waitFor` to cause a "wake-up" event to be inserted into the scheduling thread's event list, time stamped with the current time plus the `waitFor` argument. Here variable `time` is the future-event time; method `insertProcess` puts the process into the event queue. A non-Simple process (e.g., one implemented with a Java thread) goes through a sequence of synchronization steps to reach the `notify()` method. (We will say more about Simple processes in 14.2.2.) The scheduler thread has blocked on the process's native lock; this `notify()` releases it. The process then immediately calls `wait()` on its `lock` variable, which suspends the thread until the scheduler executes `notify()` on that same variable. From the point of view of the code fragment executing `waitFor`, the statement following the `waitFor` call executes precisely at the time implied by the `waitFor` argument. The code in Figure 14.2 (taken from an SSF implementation) illustrates this.

```
public void waitFor(long timeinterval){
   time = owner.owner.clock + timeinterval;
   owner.owner.insertProcess(this);
   if (!isSimple()){
      synchronized(lock){
      synchronized(this){
          notify();
        }
        try{lock.wait();}
        catch(InterruptedException e){}
      }
   }
}
```

Figure 14.2 SSF implementation of waitFor statement.

The call to `waitOn` in the **Server**'s `action` has a slightly different implementation. The code implementing `waitOn` first attaches the process to the `inChannel`'s list of processes that are blocked on it, then engages in the same lock synchronization sequence as `waitFor` to block itself and release the scheduler thread. The semantics of releasing a blocked process are defined in terms of SSF `Events`. An `outChannel` object to which an `Event` object is written has almost always been "mapped" to an `inChannel` object. When an `Event` is written to an `outChannel` at time t, the `outChannel`'s `write` method computes the time $t + d$ at which the `Event` is available on the associated `inChannel` (d is a function of delays declared when the `outChannel` is created, the `mapTo` method is called, and the `write` method is called), and an internal event is put on the scheduler's event list, with time stamp $t + d$. The scheduler executes this event (no SSF process does) and releases all processes blocked on the `inChannel` to which the `Event` arrives. Each of these is able to get a copy of the `Event` so delivered, by calling the `inChannel`'s `activeEvents` method.

From these descriptions, we see that, normally, each event has a thread overhead cost: 2 thread reanimations, and 2 thread suspensions. Depending on how thread context switching is implemented, this cost ranges from heavy to very heavy, as compared with a purely event-oriented view. These costs can be avoided in SSF by designing processes to be *simple*, as is described next.

14.2.2 Event Orientation

From a methodological point of view, the process-oriented view is distinguished from the event-oriented view in terms of the focus of the model description. Process orientation allows for a continuous description, with pauses or suspensions. Event orientation does not. From an *implementation* point of view, the key distinguishing feature of process-oriented simulation is the need to support suspension and reanimation, which leads us to threads, as we have seen. In SSF, though, we see that the difference between process and event orientation is not very large: The SSF world encompasses both. The only difference is that, for SSF to be event oriented, its processes need to be *simple*, a technical term for the case when every statement in `action` that might suspend the process would be the last statement executed under normal execution semantics.

The implementation of `waitFor` in Figure 14.2 computes the time when the suspension is lifted and puts a reanimation event in the event list. Synchronization by threads through locks is used only if the process is not *simple*. An implementation of `waitOn` would be entirely similar. If every SSF process in a model is *simple*, there is no true code suspension, and the model is essentially event oriented. The `action` body for a *simple* process is just executed from its normal entry point when the condition that releases that process from "suspension" is satisfied. The only way an **Event** that is written into an `outChannel` is delivered is if the recipient had called `waitOn` for the corresponding `inChannel` at a time prior to that at which the `Event` was written. Thus, we see that some of the "events" implicit in an SSF model with event orientation are kernel events, which decide whether model events ought to be executed as a result. Writing to an `outChannel` schedules a kernel event at the `Event`'s receive time, but the kernel's processing of that event determines whether an `action` body is called. Nevertheless, execution of `action` bodies constitutes the essential "event processing" when SSF is used in a purely event-oriented view. It is interesting that, from a conceptual point of view, there is very little difference between process-oriented and event-oriented SSF.

To conclude this discussion on tools, we remark that *flexibility* is the key requirement in computer-systems simulation. Flexibility in most contexts means the ability to use the full power of a general programming language. This requires a level of programming expertise that is not needed by users of commercial graphically oriented modeling packages. The implementation requirements of an object-oriented event-oriented approach are much less delicate than those of a threaded simulator, and the amount of simulator overhead involved in delivering an event to an object is considerably less than the cost of a context switch in a threaded system. For these reasons, most of the simulators written from scratch take the event-oriented view. However, the underlying simulation framework necessarily provides a lower level of abstraction and so forces a modeler to design and implement more model-management logic. The choice between using a process-oriented or an event-oriented simulator—or writing one's own—is a function of the level of modeling ease, versus execution speed.

To summarize this section, we present a table that lists different levels of abstraction in computer-systems simulation, the sorts of questions whose answers are sought from the models, and the sorts of tools typically used for modeling. The level of abstraction decreases as one descends through Table 14.1.

14.3 MODEL INPUT

Just as there are different levels of abstraction in computer-systems simulation, there are different means of providing input to a model. The model might be driven by stochastically generated input, or it might be given trace input, measured from actual systems. Simulations at the high end of the abstraction hierarchy most typically use stochastic input; simulations at lower levels of abstraction commonly employ trace input. Stochastic input models are particularly useful when one wishes to study system behavior over a range of scenarios; it could be that all that is required is to adjust an input model parameter and rerun the simulation. Of course, using randomly generated input raises the question of how real or representative the input is; that doubt frequently induces systems people to prefer trace data on lower level simulations. Using a trace means one cannot explore different input scenarios, but traces are useful when directly comparing two different implementations of some policy or some mechanism on the same input. The realism of the input gives the simulation added authority.

In all cases, the data used to drive the simulation is intended to exercise whatever facet of the computer system is of interest. High-level systems simulations accept a stream of job descriptions; CPU simulations accept a stream of instruction descriptions; memory simulations accept a stream of memory references; and gate-level simulations accept a stream of logical signals.

Computer systems modeled as queueing networks (recall Chapter 7) typically interpret "customers" as computer programs; servers typically represent services such as attention by the CPU or an Input–Output (I/O) system. Random sampling generates customer interarrival times; it may also be used to govern routing and time in service. However, it is common in computer-systems contexts to have routing and service times be state dependent (e.g., the next server visited is already specified in the customer's description, or could be the attached server with least queue length).

Interarrival processes have historically been modeled as Poisson processes (where times between successive arrivals have an exponential distribution). However, this assumption has fallen from favor as a result of empirical observations that significantly contradict Poisson assumptions in current computer and communication systems. The real value of Poisson assumptions lies in tractability for mathematical analysis, so, as simulationists, we can discard them with little loss.

In the subsections to follow, we look at the mathematical formulation of common input models, stochastic input models for virtual memory, and direct-execution techniques.

Table 14.1 Decreasing Abstraction and Model Results

Typical System	*Model Results*	*Tools*
CPU Network	job throughput, job response time	queueing network, Petri net simulators, scratch
Processor	instruction throughput, time/instruction	VHDL, scratch
Memory System	miss rates, response time	VHDL, scratch
ALU	timing, correctness	VHDL, scratch
Logic Network	timing, correctness	VHDL, scratch

14.3.1 Modulated Poisson Process

Stochastic input models ought to reflect the real-life phenomenon called *burstiness*—that is, brief periods when traffic intensity is much higher than normal. An input model sometimes used to support this, retaining a useful level of mathematical tractability, is a *Modulated Poisson Process*, or MPP. (See Fischer and Meier-Hellstern, [1993].) The underlying framework is a continuous-time Markov chain (CTMC), whose details we sketch so as to employ the concept later. A CTMC is always in some *state*; for descriptive purposes, states are named by the integers: 1, 2, ... The CTMC remains in a state for a random period of time, transitions randomly to another state, stays there for a random period of time, transitions again, and so on. The CTMC behavior is completely described by its *generator matrix*, $Q = \{q_{i,j}\}$. For states $i \neq j$, entry $q_{i,j}$ describes the rate at which the chain transitions from state i into state j (this is the total transition rate out of state i, times the probability that it transitions then into state j). The rate describes how quickly the transition is made; its units are transitions per unit simulation time. Diagonal element $q_{i,j}$ is the negated sum of all rates out of state i : $q_{i,i} = -\sum_{j \neq i} q_{i,j}$. An operational view of the CTMC is that, upon entering a state i, it remains in that state for an exponentially distributed period of time, the exponential having rate $-q_{i,i}$. When making the transition, it chooses state j with probability $-q_{i,j}/q_{i,i}$. Many CTMCs are *ergodic*, meaning that, if it is left to run forever, every state is visited infinitely often. In an ergodic chain, π_i denotes state i's *stationary probability*, which we can interpret as the long-term average fraction of time the CTMC is in state i. A critical relationship exists between stationary probabilities and transition rates : For every state i,

$$\pi_i \sum_{j \neq i} q_{i,j} = \sum_{j \neq i} \pi_j q_{j,i}$$

If we think of $q_{i,j}$ as describing a probability "flow" that is enabled when the CTMC is in state i, then these equations say that, in the long term, the sum of all flows out of state i is the same as the sum of all flows into the state. We will see in the example that follows that we can use the balance equations to build a stochastic input with desired characteristics. To complete the definition of a MPP, it remains only to associate a customer arrival rate λ_i with state i. When the CTMC is in state i, customers are generated as a Poisson process with rate λ_i.

To illustrate, let us consider an input process that is either OFF, ON, or BURSTY (the output rate is much higher in the BURSTY state than in the ON state). We wish for the process to be OFF half of the time—on average, for 1 second—and, when it is not OFF, we wish for it to be BURSTY for 10% of the time. We will assume that the CTMC transitions into BURSTY only from the ON state and transitions out of BURSTY only into the ON state. We will say that state 0 corresponds to OFF, 1 to ON, and 2 to BURSTY. Our problem statement implies that $\pi_0 = 0.5$, $\pi_1 = 0.45$, and $\pi_2 = 0.05$. The only transition from OFF is to ON, and the mean OFF time is 1, so we infer that $q_{0,1} = 1$. The balance equation for state 0 can be rewritten as

$$0.5 = 0.45 q_{1,0}$$

and hence $q_{1,0} = (0.5/0.45)$. The balance equation for state 1 can be rewritten as

$$0.45((0.5/0.45) + q_{1,2}) = 0.5 + 0.05 q_{2,1}$$

and the balance equation for state 2 is

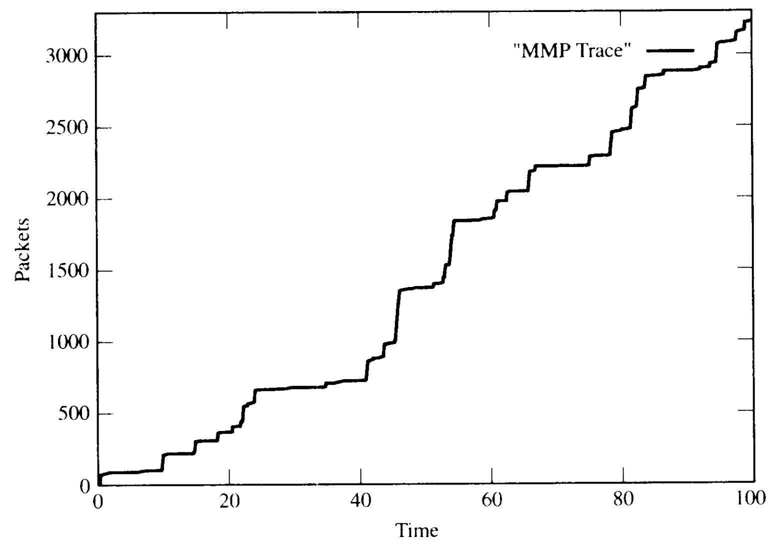
$$0.05 q_{2,1} = 0.45 q_{1,2}$$

The equations for states 1 and 2 are identical; mathematically, we don't have enough conditions to force a unique solution. If we add the constraint that a BURSTY period lasts, on average, 1/10 of a second, we thereby define that $q_{2,1} = 10$ and, hence, that $q_{1,2} = (0.5/0.45)$. Operationally, the simulation of this CTMC is straightforward. In state 0, one samples an exponential with mean 1 to determine the state's holding time. Following this period, the CTMC transitions into state 1 and samples a holding time from an exponential with mean 0.45, after which it transitions to OFF or BURSTY with equal probability. In the BURSTY state, it samples an exponential holding time with mean 0.1. Now all that is left is for us to define the state-dependent customer arrival rates. Obviously, $\lambda_0 = 0$; for illustration, we choose $\lambda_1 = 10$ and $\lambda_2 = 500$.

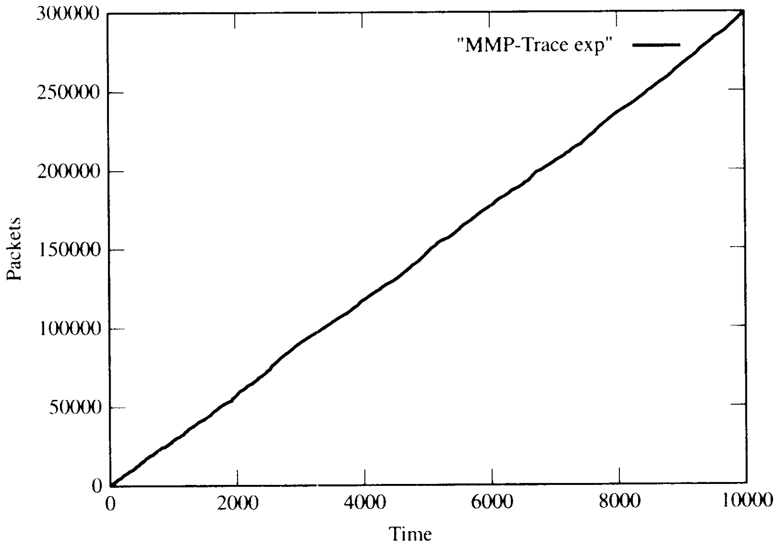
Figure 14.3 presents a snippet of code used to generate times of arrivals in this process. Transitions between states are sampled by using the inverse-transform technique, described in Chapter 9. (The variable `acc` computes the cumulative probability function in the distribution described by the row vector `P[state]`.) Figure 14.4 plots total customers generated as a function of time—for a short period of a sample run, and for a longer period. In the shorter run, we see regions where the graph increases sharply; they correspond to periods in the BURSTY state. While the CTMC is not in this state, a mixture of OFF and ON periods moves the accumulated packet count up at a much more gradual rate. The MPP model can describe burstiness, but the burstiness is limited in time scale. The longer run views the data at a time scale that is two orders of magnitude larger, and we see that the irregularities are largely smoothed.

```
class mpp {

 public static double Finish;          // sim termination
 public static double time = 0.0;      // current clock
 public static double htime, etime;    // transition times
 public static int state = 0;          // current state id
 public static int total = 0;          // total pkts emitted
 public static Random stream;
  ...

public static void main(String argv[]) {
  ...
while( time < Finish ) {

 // generate exponential holding time, state-dependent mean
 htime = time+exponential( stream, hold[state] );

 // emit packets until state transition time. State dependent
 // rate. Note assignment made to etime in while condition test
 while( (etime = time+exponential( stream, 1.0/rate[state]))
                    < min( htime, Finish) ) {
        System.out.println( etime + '' '' + total);
    total++;
    time = etime; // advance to packet issue time
 }
 time = htime;

 // select next state
 double trans = stream.nextDouble();
 double acc = P[state][0];
 int i = 0;

 while( acc < trans ) acc += P[state][++i];
 state = i;
  }
 }
  ...
}
```

Figure 14.3 Java code generating MPP trace.



(a) Short run, small time scale



(b) Long run, large time scale

Figure 14.4 Sample runs from MPP model.

In contrast to the Markovian essence of the MPP model, consider a traffic source that remains OFF for an exponentially distributed period of time with mean 1.0, but, when it comes ON, remains on for a period of time sampled from a Pareto distribution. While it is ON, packets arrive as a Poisson process. As we will see in the chapter on simulation of computer networks, the Pareto distribution is of particular interest because it gives rise to "self-similarity," which informally means preservation of irregularities at multiple time scales. Figure 14.5 parallels the MPP data, displaying accumulated packet counts as a function of time; it presents behavior for the first 1000 units of time and for the first 100,000 units of time. Here, despite two orders of

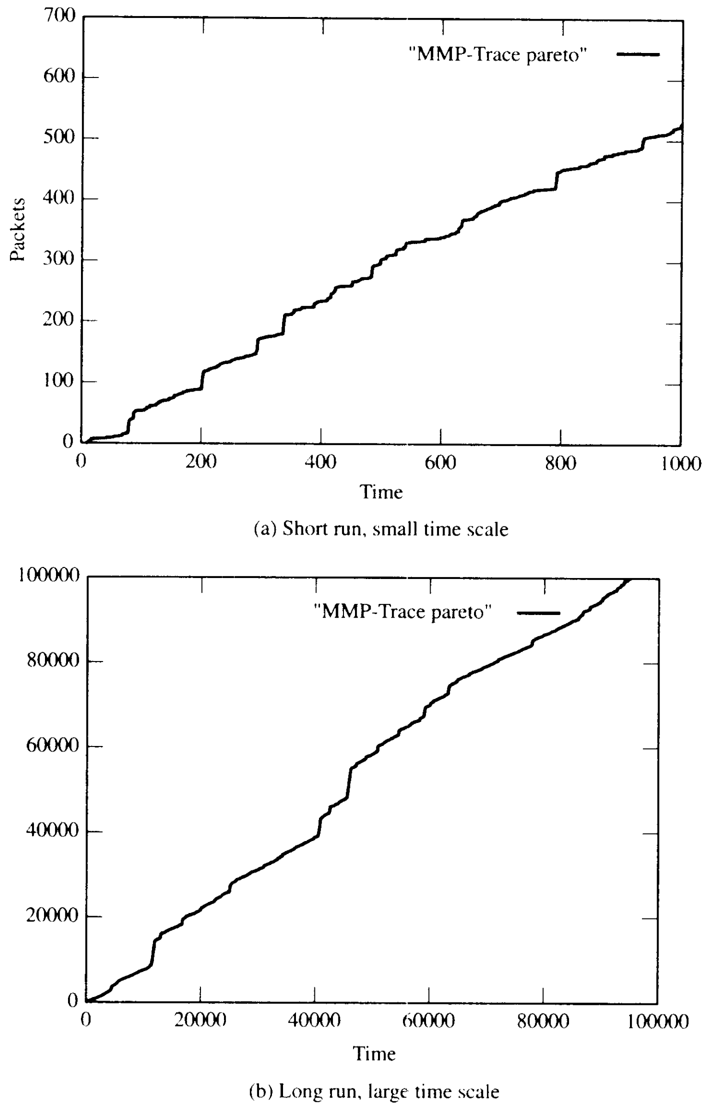


Figure 14.5 Sample runs from self-similar model.

magnitude of difference in run length, the visual impression of behavior is much the same between the two traces. This sort of behavior is frequently seen in computer and communication systems; the long lengths reflect burstiness of packets, file lengths, and demand on a server.

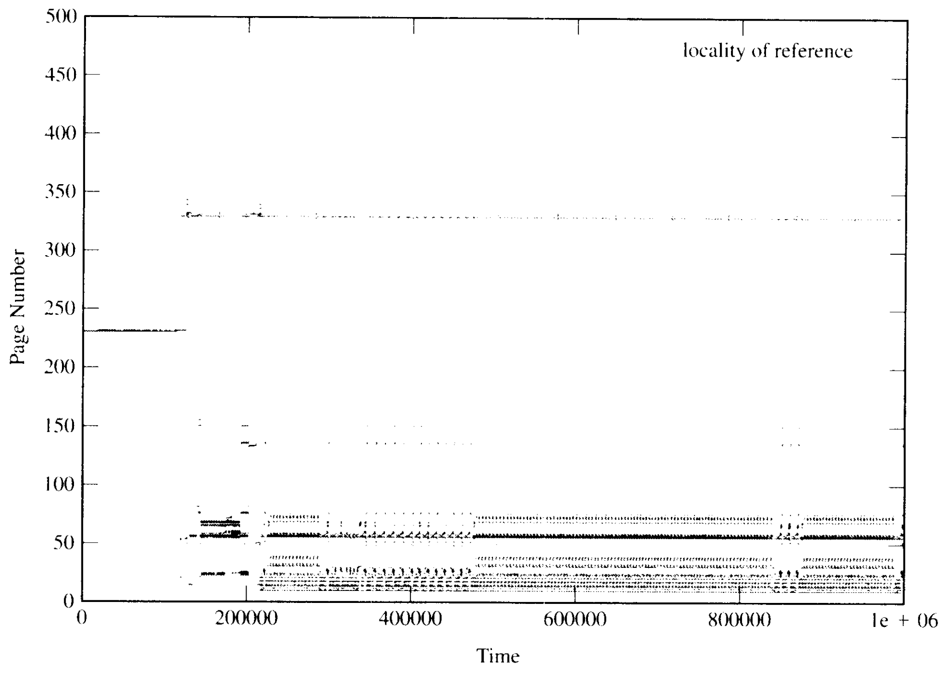
14.3.2 Virtual-Memory Referencing

Randomness can also be used to drive models in the middle levels of abstraction. An example is a model of program-execution behavior in a computer with virtual memory. (See Nutt [2004].) In such a system, the data and instructions used by the program are organized in units called *pages*. All pages are the same size, typically 2^{10} to 2^{12} bytes in size. The physical memory of a computer is divided into *page frames*, each capable of holding exactly one page. The decision of which page to map to which frame is made by the operating system. As the program executes, it makes memory references to the "virtual memory," as if it occupied a very large memory starting at address 0 and were the only occupant of the memory. On every memory reference made by the program, the hardware looks up the identity of the page frame containing the reference and translates the virtual address into a physical address. The hardware might discover that the referenced page is not present in the main memory; this situation is called a *page fault*. When a page fault occurs, the hardware alerts the operating system, which then takes over to bring in the referenced page from a disk and decides which page frame should contain it. The operating system could need to evict a page from a page frame to make room for the new one. The policy the operating system uses to decide which page to evict is called the "replacement policy." The quality of a replacement policy is often measured in terms of the *hit ratio*—the fraction of references made whose page frames are found immediately.

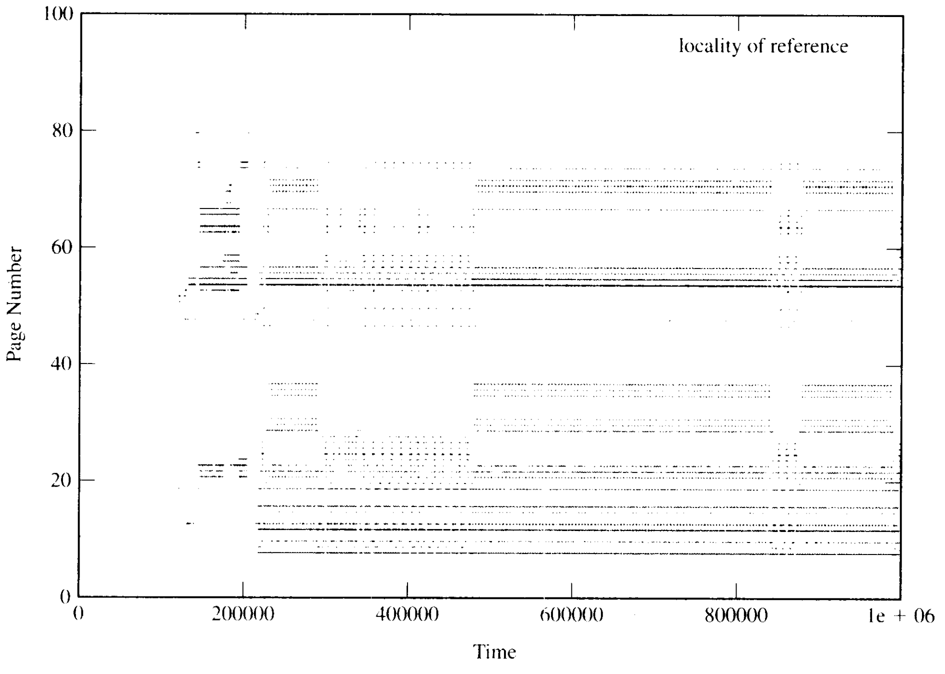
Virtual-memory systems are used in computers that support concurrent execution of multiple programs. In order to study different replacement policies, one could simulate the memory-referencing behavior of several different programs, simulate the replacement policy, and count the number of references that page fault. For this simulation to be meaningful, it is necessary that the stochastically generated references capture essential characteristics of program behavior. Virtual memory works well precisely because programs do tend to exhibit a certain type of behavior; this behavior is called locality of reference. What this means intuitively is that program references tend to cluster in time and space and that, when a reference to a new page is made and the page is brought in from the disk, it is likely that the other data or instructions on the page will also soon be referenced. In this way, the overhead of bringing in the page is amortized over all the references made to that page before it is eventually evicted. A program's referencing behavior can usually be separated into a sequence of "phases"; during each phase, the program makes references to a relatively small collection of pages, called its *working set*. Phase transitions essentially change the program's working set. The challenge for the operating system is to recognize when the pages used by a program are no longer in its working set, for these are the pages it can safely evict to make room for pages that *are* in some program's working set.

Figure 14.6 illustrates a stream of memory references taken from an execution of the commonly used gcc compiler. One graph gives a global picture; the other cuts out references to pages over number 100 and shows more fine detail. Each graph depicts points of the form (i, p_i) where p_i is the page number of the ith reference made by the program (arithmetically shifted so that the smallest page number referenced is 10). The phases are clearly seen; each member of the working set of a phase is seen as lines (which are really just a concatenation of many points). One striking facet of this graph is how certain pages remain in almost all working sets. However, other kinds of programs exhibit other behaviors. A common characteristic of scientific programs is that the execution is dominated by an inner loop that sweeps over arrays of data; the pages containing the instructions are in the working set throughout the loop, but data pages migrate in and out.

Despite various differences, a near-invariant among program executions is the presence of phase-like behavior and of working sets. In the building of a stochastic reference generator, it therefore makes sense to focus modeling effort on phase and working-set definition. As a starting point, we might, with every reference generated, randomly choose (with some small probability) whether to start a new phase (by changing the working set). Given a working set, we would choose to reference some page in the working set with high probability and, if choosing to stay in the set, choose with high probability the same page as the one last referenced in the working set. The inner loop of a program that generates references in this fashion appears in Figure 14.7. Details of working-set definition are hidden inside of routine new_wrkset and might vary with the type of program being modeled. For the purposes of illustration here, we wrote a version that defined a working set by randomly choosing a working-set size between 2 and 8 and a maximum page number of 100. A working set of size n is constructed by randomly choosing a "center" page c from among all pages, randomly choosing an integer dispersion factor d from 2 to 6, and then randomly selecting a working set from among all pages within distance $d \times n$ from center page c (with appropriate wraparound of page numbers at the endpoints 0 and 100). In order to model the referencing pattern of a scientific program's



(a) gcc, all references shown



(b) gcc, references < 100 shown

Figure 14.6 Scatter-plotted referencing pattern of gcc compiler. Referenced page number is plotted as a function of reference number ("time"). Horizontal sequences indicate frequent rereferences to the same page number.

```
double ppt = 0.0001;   // Pr{phase transition}
double psw = 0.999;    // Pr{ref in WS}
double psp = 0.9;      // Pr{reference same page}

// method new_wrkset() creates a new working set
// method from_wrkset() samples from the working set
// method not_from_wrkset() samples from outside the working set

int ref;               // last page referenced
int sv_ref;            // save ref
Random stream;         // random number stream

...

for(int i=0; i<length; i++) {
   if( stream.nextDouble() < ppt ) new_wrkset();    // phase transition
   if( stream.nextDouble() < psw ) {                 // stay in working set?
    if( psp < stream.nextDouble() )                      // change page, in wrkset
      ref = sv_ref = from_wrkset();
  } else ref = not_from_wrkset();    // step outside of wrkset

System.out.println(i + `` '' + ref);
ref = sv_ref;
}
```

Figure 14.7 Java pseudocode for generating a reference trace.

instruction stream, we manipulated the logic illustrated above to "lock down" a working set for a long time in the middle of the program execution. Figure 14.8 illustrates the result. As designed, phases and working sets are precisely defined.

The preceding example illustrates how one can in principle generate an execution path stochastically, but simulations at the middle level of abstraction also commonly use traces. Studies of CPU design will use a measured trace of instructions executed by a running program; studies of memory systems will use a measured trace of the addresses referenced by an executing program. Such traces get to be lengthy. A small piece of a typical trace of memory references is shown here:

```
2     430d70
2     430d74
2     415130
0   1000acac
2     414134
1   7fff00ac
2     414138
```

The first number is a code describing the type of access; 2 represents an instruction fetch, 0 a data read, 1 a data write. The second number represents a memory address, in hexadecimal. If the trace were also to describe the instruction stream, a hexadecimal word giving the machine code of the instruction fetched could follow the memory address on every instruction fetch line. Two or three words of memory are needed to represent one reference, even when the information is efficiently packed (not as characters, as shown, which take much more space!). Consider also the amount of computation needed to simulate a CPU or memory for the execution of a significantly long run of a nontrivial program. These observations help us understand the motivation for techniques that compress the address trace and for techniques that allow one to infer information about multiple systems from a single pass through a long trace. We will say more about these techniques later in this chapter.

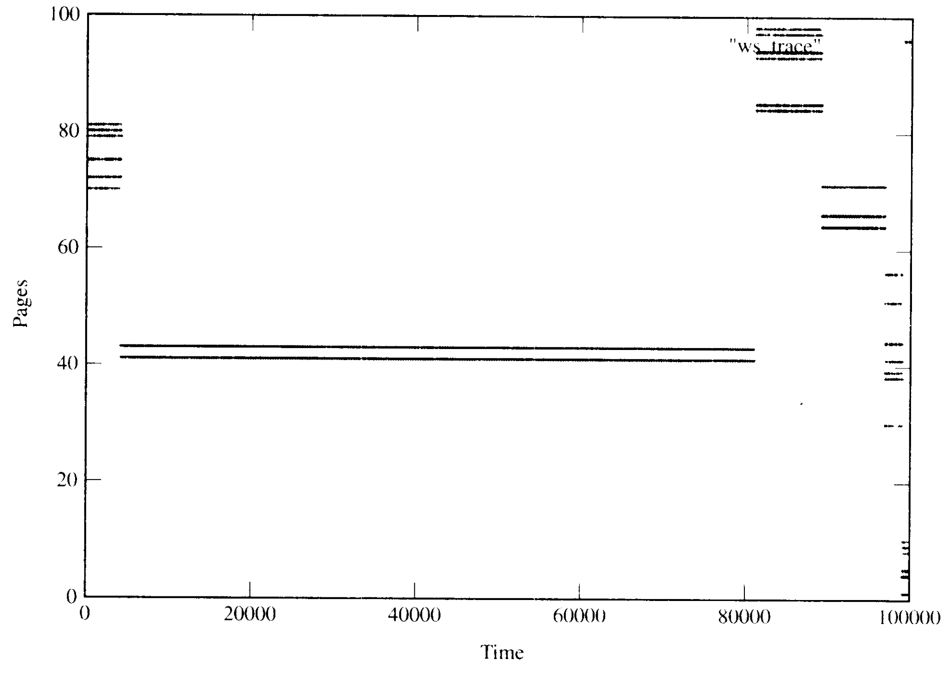


Figure 14.8 A synthetic trace modeling a scientific-program instruction stream.

Another method of generating input is called "direct execution" simulation. (For examples, see Covington *et al.* [1991], Lebeck and Wood [1997], Dickens *et al.* [1996]). One approach to it is illustrated in Figure 14.9. Direct execution is like generating a trace and driving the simulation with that trace, all at once. Computer programs are "instrumented" with additional code that observes the instructions the program executes and the

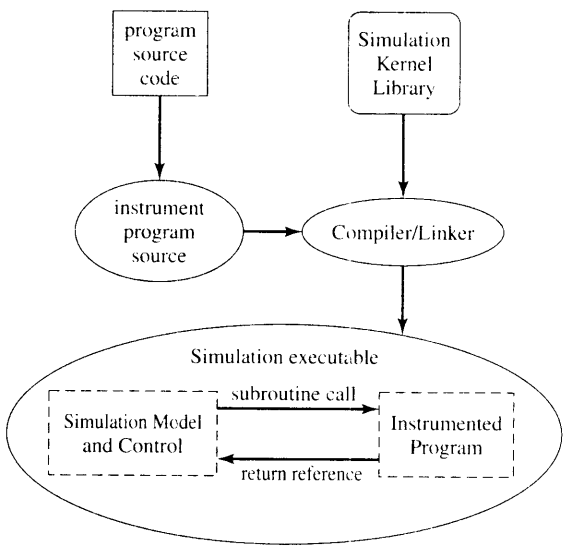


Figure 14.9 Direct-execution simulation.

memory and I/O references the program makes as it executes. The instrumented program is compiled and linked with a simulation kernel library. Execution control rests with the simulation kernel, which calls the instrumented program to provide the next instruction or reference that the program generates. The simulation kernel uses the returned information to drive the model for the next step. The simulation model driven by the program's execution can be of an entirely different CPU design, or a memory system, or even (given multiple instrumented programs) the internals of a communications network. Direct-execution simulation solves the problem of storing very large traces—the trace is consumed as it is being generated. However, it is tricky to modify computer programs to get at the trace information and to coordinate the trace generator with discrete-event simulator. The only practical way an ordinary simulator practitioner can use such methods is when the system has a software tool for making such modifications, but this feature is not common.

14.4 HIGH-LEVEL COMPUTER-SYSTEM SIMULATION

In this section, we illustrate concepts typical of high-level computer simulations by sketching a simulation model of a computer system that services requests from the World Wide Web.

Example 14.1

A company that provides a major website for searching and links to sites for travel, commerce, entertainment, and the like wishes to conduct a capacity-planning study. The overall architecture of its system is shown in Figure 14.10. At the back end, one finds data servers responsible for all aspects of handling specific queries and updating databases. Data servers receive requests for service from application servers—machines dedicated to running specific applications (e.g., a search engine) supported by the site. In front of the applications are Web servers, which manage the interaction of applications with the World Wide Web; the portal to the whole system is a load-balancing router that distributes requests directed to the website among the Web servers.

The goal of the study is to evaluate the site's ability to handle load at peak periods. The desired output is an empirical distribution of the access response time. Thus, the high-level simulation model should focus

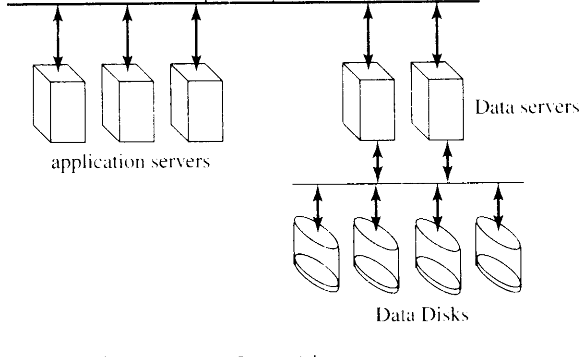


Figure 14.10 Website server system.

on the impact of timing at each level that is used, system factors that affect that timing, and the effects of timing on contention for resources. To understand where those delays occur, let us consider the processing associated with a typical query.

All entries into the system are through a dedicated router, which examines the request and forwards it to some Web server. Time is required to exercise the logic of looking at the request to discern whether it is a new request (requiring load balancing) or part of an ongoing session. It is reasonable to assume one switching time for a preexisting request and a different time for a new request. The result of the first step is selection of a Web server and the enqueueing there of a request for service. A Web server can be thought of as having one queue of threads of new requests, a second queue of threads that are suspended awaiting a response from an application server, and a third queue of threads "ready" to process responses from application servers. An accepted request from the router creates a new request thread. We may assume the Web server has adequate memory to deal with all requests. It has a queueing policy that manages access to the CPU; the distinction between new requests and responses from application servers is maintained for the sake of scheduling and for the sake of assigning service times, the distributions of which depend on the type. The servicing of a new request amounts to identification of an application and the associated application server. A request for service is formatted and forwarded to an application server, and the requesting thread joins the suspended queue. At an application server, requests for service are organized along application types. A new request creates a thread that joins a new-request queue associated with the identified application. An application request is modeled as a sequence of sets of requests from data servers, interspersed with computational bursts—for example,

```
burst 1
request data from D1, D3, and D5
burst 2
request data from D1 and D2
burst 3
```

In this model, we assume that all data requests from a set must be satisfied before the subsequent computational burst can begin. Query search on a database is an example of an application that could generate a long sequence of bursts and data requests, with large numbers of data requests in each set. We need not assume that every execution of an application is identical in terms of data requests or execution bursts; these can be generated stochastically. An application thread's state will include description of its location in its sequence and a list of data requests still outstanding before the thread can execute again. Thus, for each application, we will maintain a list of threads that are ready to execute and a list of threads that are suspended awaiting responses from data servers. An application server will implement a scheduling policy over sets of ready application threads. A data server creates a new thread to respond to a data request and places it in a queue of ready threads. Some data server might implement memory-management policies and could require further coordination with the application server to know when to release used memory. Upon receiving service, the thread requests data from a disk, then suspends until the disk operation completes, at which point the thread is moved from the suspended list to the ready list and, when executed, again reports back to the application server associated with the request. The thread suspended at the application server responds; eventually, the application thread finishes and reports its completion back to the Web-server thread that initiated it, which in turn communicates the results back over the Internet.

Stepping back from the details, we see that a simulation model of this system must specify a number of features, listed in Table 14.2. All of these affecting timing in some way. The query-response-time distribution can be estimated by measuring, for each query, the time between at which a request first hits the router and the time at which the Web-server thread communicates the results. From the set of simulated queries, one can build up a histogram. As should be evident, a response time reflects a great many different factors related to execution bursts, scheduling policies, and disk-access times. Deeper understanding of the system is obtained by measuring behavior at each server of each type. One would look especially for evidence of

Table 14.2 Required Specification for Web System Model

Subsystem	*Specifications*
Router	load-balancing policy, execution times
Web Server	server count, queueing policy, execution times
Application Server	server count, queueing policy, behavior model
Data Server	server count, disk count, queueing policy, memory policy, disk timing

bottlenecks. CPU bottlenecks would be reflected at servers with high CPU utilization; IO bottlenecks at disks with high utilization. To assess system capacity at peak loads, we would simulate to identify bottlenecks, then look to see how to reduce load at bottleneck devices by changes in scheduling policies, by binding of applications to servers, or by increasing the number of CPUs or disks in the system. Normally, one must resimulate a reconfigured system under the same load as before to assess the effects of the changes.

The website model is an excellent candidate for a threaded (process-oriented) approach to modeling. The most natural process-oriented approach is to associate processes with servers. The simulation model is expressed from an abstracted point of view of the servers' operating system. Individual queries become messages that are passed between server processes. In additional to limiting the number of processes, an advantage of this approach is that it explicitly exposes the scheduling of query processing at the user level. The modeler has both the opportunity and the responsibility to provide the logic of scheduling actions that model processing done on behalf of a query. It is a modeling viewpoint that simplifies analysis of server behavior—an overloaded server is easily identified by the (modeler-observable) length of its queue of runnable queries. However, it is a modeling viewpoint that is a bit lower in abstraction than the first one and requires more modeling and coding on the part of the user.

An event-oriented model of this system need not look a great deal different from the second of our process-oriented models. A query passed as a message between servers have an obvious event-oriented expression. A modeler would have to add to the logic, events, and event handlers that describe the way a CPU passes through simulation time. For example, consider a call to *hold(qt)* in a process-oriented model to express that the CPU is allocating `qt` units of service to a query, during which time it does nothing else. In an event-oriented model, one would need to define events that reflect "starting" and "stopping" the processing of a query, with some scheduling logic interspersed. Additional events and handlers need to be defined for any "signaling" that might be done between servers in a process-oriented model—for example, when a data-server process awaits completion of modeled IO requests sent to its disks. A process-oriented approach, even one focused on servers rather than queries, lifts the level of model expression to a higher level of abstraction and reduces the amount of code that must be written. In a system as complex as the website, one must factor complexity of expression into the overall model-development process.

14.5 CPU SIMULATION

Next, we consider a lower level of abstraction and look at the simulation of a central processing unit. Whereas the high-level simulation of the previous example treated execution time of a program as a constant, at the lower level we do the simulation to discover what the execution time is. The input driving this simulation is a stream of instructions. The simulation works through the mechanics of the CPU's logical design to find out what happens in response to that stream, how long it takes to execute the program, and where bottlenecks exist in the CPU design. Our discussion illustrates some of the functionality of a modern CPU and the model characteristics that such a simulation seeks to discern. Examples of such simulations include those described in Cmelik and Keppel [1994], Bedicheck [1995], Witchel and Rosenblum [1996], Austin, Larson, and Ernst [2002], Bohrer *et al.* [2004] and Magnusson *et al.* [2002]. The view of the CPU taken in our discussion is similar to that taken by the RSIM system (Hughes *et al.* [2002]).

The main challenge to making effective use of a CPU is to avoid stalling it; stalling happens whenever the CPU commits to executing an instruction whose inputs are not all present. A leading cause of stalls is the latency delay between CPU and main memory, which can be tens of CPU cycles. One instruction might initiate a read—for example,

```
load $2, 4($3)
```

which is an assembly language statement that instructs the CPU to use the data in register 3 (after adding value 4 to it) as a memory address and to put the data found at that address into register 2. If the CPU insisted on waiting for that data to appear in register 2 before further execution, the instruction could stall the CPU for a long time if the referenced address is not found in the cache. High-performance CPUs avoid this by recognizing that additional instructions *can* be executed, up to the point where the CPU attempts to execute an instruction that reads the contents of register 2—for example,

```
add $4,$2,$5
```

This instruction adds the contents of registers 2 and 5, and places the result in register 4. If the data expected in register 2 is not yet present, the CPU will stall. So we see that, to allow the CPU to continue past a memory load, it is necessary to (1) mark the target register as being unready, (2) allow the memory system to load the target register asynchronously while the CPU continues on in the instruction stream, (3) stall the CPU if it attempts to read a register marked as unready, and (4) clear the unready status when the memory operation completes.

The sort of arrangement just described was first used in the earliest supercomputers, designed in the 1960s. Modern microprocessors add some additional capabilities to exploit *instruction level parallelism* (ILP). We outline some of the current architecture ideas in use to illustrate what a simulation model of an ILP CPU involves.

The technique of pipelining has long been recognized as a way of accelerating the execution of computer instructions. (See Patterson and Hennessy [1997].) Pipelining exploits the fact that each instruction goes sequentially through several stages in the course of being processed; separate hardware resources are dedicated to each stage, permitting multiple instructions to be in various stages of processing concurrently. A typical sequence of stages in an ILP CPU is as follows:

1. *Instruction fetch*: The instruction is fetched from the memory.
2. *Instruction decode*: The memory word holding the instruction is interpreted to discover what operation is specified; the registers involved are identified.
3. *Instruction issue*: An instruction is "issued" when there are no constraints holding it back from being executed. Constraints that keep an instruction from being issued include data not yet being ready in an input register and unavailability of a functional unit (e.g., Arithmetic Logical Unit) needed to execute the instruction.
4. *Instruction execute*: The instruction is performed.
5. *Instruction complete*: Results of the instruction are stored in the destination register.
6. *Instruction graduate*: Executed instructions are graduated in the order that they appear in the instruction stream.

Ordinary pipelines permit at most one instruction to be represented in each stage; the degree of parallelism (number of concurrent instructions) is limited to the number of stages. ILP designs allow multiple instructions to be represented in some stages. This necessarily implies the possibility of executing some stages of successively fetched instructions out of order. For example, it is entirely possible for the n^{th} instruction, I_n, to be constrained from being issued for several clock cycles while the next instruction, I_{n+1}, is not so constrained. An ILP processor will push the evaluation of I_{n+1} along as far as it can without waiting on I_n. However, the *instruction graduate* stage will reimpose order and insist on graduating I_n before I_{n+1}.

ILP CPUs use architectural slight of hand with respect to register useage to accelerate performance. An ILP machine typically has more registers available than appear in the instruction set. Registers named in instructions need not precisely be the registers actually used in the implementation of those instructions. This is acceptable, of course, as long as the effect of the instructions is the same in the end. One factor motivating this design is the possibility of having multiple instructions involving the same logical registers (those named by the instructions themselves) actively being processed concurrently. By providing each instruction with its own "copy" of a register, we eliminate one source of stalls. Another factor involves branches—that is, instructions that interrupt the sequential flow of control. An ILP, encountering a branch instruction, will predict whether the branch is taken or not and possibly alter the instruction stream as a result. Various methods exist to predict branching, but any of them will occasionally predict incorrectly. When an incorrect prediction is made, the register state computed as a result of speculating on branch outcome needs to be discarded and execution resumed at the branch point. Thus, another use of additional registers is to store the "speculative register state." With dedicated hardware resources to track register useage following speculative branch decision, speculative state can be discarded in a single cycle and control resumed at the mispredicted branch point. In all of these cases, the hardware implements techniques for renaming the logical registers that appear in the instructions to physical registers, for maintaining the mapping of logical to physical registers, and for managing physical register useage.

A simulation model of an ILP CPU will model the logic of each stage and coordinate the movement of instructions from stage to stage. We consider each stage in turn.

An instruction-fetch stage could interact with the simulated memory system, if that is present. However, if the CPU simulation is driven by a direct-execution simulation or by a trace file, there is little for a model of this stage to do but get the next instruction in the stream. If a memory system is present, this stage could look into an instruction cache for the next referenced instruction, stalling if a miss is suffered.

Following an instruction fetch, an instruction will be in the CPU's list of active instructions until it exits altogether from the pipeline. The instruction-decode stage places an instruction in this list; a logical register that appears as the target of an operation is assigned a physical register—registers used as operand sources will have been assigned physical registers in instructions that defined their values. (Sequencing issues associated with having multiple representations of the same register are dealt with at a later stage in the pipeline.) Branch instructions are identified in this stage, predictions of branch outcomes are made, and resources for tracking speculative execution are committed here.

Decoded instructions pass into the instruction-issue stage. The logic here is complex and very much timing dependent. An instruction cannot be issued until values in its input registers are available and a functional unit needed to perform the instruction is available. An input value might be not yet in a register, for instance, if that value is loaded from memory by a previous instruction and has not yet appeared. A functional unit could be unavailable because all appropriate ones are busy with multicycle operations initiated by other instructions. Implementation of the issue-stage model (and hardware) depends on marking registers and functional units as busy or pending and on making sure that, when the state of a register or functional unit changes, any instruction that cannot yet issue because of that register or functional unit is reconsidered for issue.

Simulation of the instruction-execute stage is a matter of computing the result specified by the instruction (e.g., an addition). At this point, the action of depositing the result into a register or memory is scheduled for the instruction-complete stage. This latter stage also cleans up the status bits associated with registers and functional units involved in the instruction and resolves the final outcome of a predicted branch. If a branch was mispredicted, the speculatively fetched and processed instructions that follow it are removed from other pipeline stages, the hardware that tracks speculative instruction is released, and the instruction stream is reset to follow the branch's other decision direction.

Between the instruction-issue and instruction-complete stages, instructions could get processed in an order that does not correspond to the original instruction stream. The last stage, graduation, reorders them. Architecturally, this permits an ILP CPU to associate an exception (e.g., a page fault or a division by zero) with the precise instruction that caused it. Simulation of this stage is a matter of knowing the sequence number of the next instruction to be graduated, then graduating it when it appears.

Example 14.2

An example helps to show what goes on. Consider the following sequence of assembly-language instructions for a hypothetical computer:

```
load $2, 0($6) ; I1-  load $2 from memory
mult $5, 2     ; I2-  multiply $5 with constant 2
add  $4, 12    ; I3-  add constant 12 to $4
add  $5, $2    ; I4-  $5 <- $5 + $2
add  $5, $4    ; I5-  $5 <- $5 + $4
```

Let us suppose that the register load misses the first-level cache but hits in the second-level cache, resulting in a delay of 4 cycles before the register gets the value. Suppose further that separate hardware exists for addition and multiplication, that addition takes one cycle, and that multiplication takes 2 cycles to complete. Time is assumed to advance in units of a single clock tick.

Table 14.3 shows a timeline of when each instruction is in each stage. Cycles in which an instruction cannot proceed through the pipeline are marked as "stall" cycles. Processing is most easily understood by tracing individual instructions through.

I1. After being fetched in cycle 1, the decode of I1 assigns physical register \$p1 as the target of the load operation and marks \$p1 as unready. No constraints prohibit I1 from being issued in cycle 3 nor executed in cycle 4. Because the memory operation takes 4 cycles to finish, I1 is stalled in cycles 5–8. Cycle 9 commits the data from memory to physical register \$p1 and clears its unready flag; the instruction is graduated in cycle 10.

I2. Instruction I2 is fetched in cycle 2 and has physical register \$p2 allocated to receive the results of the multiplication in the cycle-3 decode stage; \$p2's unready flag is raised. No constraints keep I2 from

Table 14.3 Pipeline Stages, ILP CPU Simulation

Inst./Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	fetch	decode	issue	execute	stall	stall	stall	stall	complete	graduate				
I2		fetch	decode	issue	execute	stall	complete	stall	stall	stall	graduate			
I3			fetch	decode	issue	execute	complete	stall	stall	stall	stall	graduate		
I4				fetch	decode	stall	stall	stall	stall	issue	execute	complete	graduate	
I5					fetch	decode	stall	stall	stall	stall	stall	stall	issue	complete

being issued in cycle 4 or executed in cycle 5, but the 2-cycle delay of the multiplier means the result is not committed to register \$p2 until cycle 7, at which point the \$p2 unready flag is cleared. The instruction remains stalled through cycles 8–10, awaiting the graduation of I1.

I3. Instruction I3 is fetched in cycle 3 and has physical register \$p3 allocated to receive the results of its addition in the cycle-4 decode stage. The \$p3 unready flag is raised. There are no constraints keeping I3 from being issued in cycle 5 and executed in cycle 6, with results written into \$p3 in cycle 7, at which time \$p3's unready flag is cleared. I3 must stall, however, during cycles 8–11, awaiting the graduation of I2.

I4. Instruction I4 is fetched in cycle 4 and has physical register \$p4 allocated to receive the results of the addition during the cycle-5 decode stage. \$p4's unready flag is raised at that point. Physical registers \$p1 and \$p2 are operands to the addition; I4 stalls in cycles 6–9, waiting for their unready flags to clear. It then passes the remaining stages without further delay, clearing the \$p4 unready flag in cycle 12.

I5. Instruction I5 is fetched in cycle 5 and has physical register \$p5 allocated to receive the results of its addition in the cycle-6 decode stage, at which point the \$p5 unready flag is set. Physical registers \$p3 and \$p4 contain the addition's operands; I5 stalls through cycles 7–12, waiting for their unready flags both to clear. From that point forward, I5 passes through the remaining stages without further delay.

The performance benefit of pipelining and ILP can be appreciated if we compare the execution time of this sequence on a nonpipelined, non-ILP machine. Assuming that each stage must be performed for each instruction but that one instruction is processed in its entirety before another one begins, 51 cycles are needed to execute I1 through I5. With the advanced architectural features, only 15 cycles are needed. The example illustrates both the parallelism that pipelining exposes and the latency tolerance that the ILP design supports. Even though I1 stalls for four cycles while awaiting a result from memory, the pipeline keeps moving other instructions through to some extent. The bottom line for someone using a model like this is the rate at which instructions are graduated, as this reflects the effectiveness of the CPU design. Secondary statistics would try to pinpoint where in the design stalls occur that might be alleviated (e.g., if many stalls occur because of waiting for the multiplier (no such stalls occur in the example), then one could consider including an additional multiplier in the CPU design).

Our explanation of the model's workings was decidedly process oriented, taking the view of an instruction. However, the computational demands of a model like this are enormous, owing to the very large number of instructions that must be simulated to assess the CPU design on, say, a single program run. The relatively high cost of context switching would deter use of a normal process-oriented language. One could implement what is essentially a process-oriented view by using events—each time an instruction passes through a stage, an event is scheduled to take that instruction through the next stage, accounting for stalls. The amount of simulation work accomplished per event is thus the amount of work done on behalf of one instruction in one stage. An alternative approach is to eschew explicit events altogether and simply use a cycle-by-cycle activity scan. At each cycle, one would examine each active instruction to see whether any activity associated with that instruction can be done. An instruction that was at one stage at cycle j will, at cycle $j + 1$, be examined for constraints that would keep it at cycle j. Finding none, that instruction would be advanced to the next stage. An activity-scanning approach has the attractiveness of eliminating event-list overhead, but the disadvantage of expending computational effect on checking the status of a stalled instruction on every cycle during which it is stalled. Implementation details and model behavior largely determine whether an activity-scanning approach is faster than an event-oriented approach (with the nod going to activity scanning when few instructions stall).

14.6 MEMORY SIMULATION

One of the great challenges of computer architecture is finding ways to deal effectively with the increasing gap in operation speed between CPUs and main memory. A factor of 100 in speed is not far from the mark.

The main technique that has evolved is to build hierarchies of memories. A relatively small memory—the L1 cache—operates at CPU speed. A larger memory—the L2 cache—is larger and operates more slowly. The main memory is larger still and slower still. The smaller memories hold data that was referenced recently and nearby data that one hopes will also be referenced soon. Data moves up the hierarchy on demand and ages out as it becomes disused, to make room for the data in current use. For instance, when the CPU wishes to read memory location 100,000, hardware will look for it in the L1 cache; if it fails to find it there, it will look in the L2 cache. If it is found there, an entire block containing that reference is moved from the L2 cache into the L1 cache. If it is not found in the L2 cache, a (larger) block of data containing location 10,000 is copied from the main memory to the L2 cache, and part of that block (containing location 10,000 of course) is copied into the L1 cache. It could take 50 cycles or more to accomplish this. After this cost has been suffered, the hope and expectation is that the CPU will continue to make references to data in the block brought in, because accesses to L1 data are made at CPU speeds. Fortunately, most programs exhibit locality of reference at this scale (as well as at the paging scale discussed earlier in the chapter), so the strategy works. However, after a block ceases to be referenced for a time, it is ejected from the L1 cache. It could remain in the L2 cache for a while and later be brought back into the L1 cache if any element of the block is referenced again. Eventually a block remains unreferenced long enough so that it is ejected also from the L2 cache.

The astute reader will realize that data that is written into an L1 cache by the CPU creates a consistency problem, in that a memory address then has different values associated with it at different levels of the memory hierarchy. One way of dealing with this is to write through to all cache levels every time there is a write—the new value is asynchronously pushed from L1 through L2 to the main memory. An alternative method copies back a block from one memory level to the lower level, at the point the block is being ejected from the faster level. The write-through strategy avoids writing back blocks when they are ejected, whereas the write-back strategy requires that an entire block be written back when ejected, even if only one word of the block was modified, once. One of the roles simulation plays is to compare performance of these two write-back strategies, taking into consideration all costs and contention for the resources needed to support writing back modifications.

Like paging systems, the principle measure of the quality of a memory hierarchy is its hit ratio at each level. As with CPU models, to evaluate a memory hierarchy design, one must study the design in response to a very long string of memory references. Direct-execution simulation can provide such a reference stream, as can long traces of measured reference traffic. Nearly every caching system is a demand system, which means that a new block is not brought into a cache before a reference is made to a word in that block. Decisions left still to the designer include whether to write-through or write-back modifications, the replacement policy, and the "set associativity."

The concept of set associativity arises in response to the cost of the mechanism used to look for a match. Imagine we have an L2 cache with 2 million memory words (an actual figure from an actual machine). The CPU references location 10000—the main memory has, say, 2^{12} words, so the L2 cache holds but a minute fraction of the memory. How does the hardware find out whether location 10000 is in the L2 cache? It uses what is called an associative memory, one that associates search keys with data. One queries an associative memory by providing some search key. If the key is found in the memory, then the data associated with the key is returned; otherwise, indication of failure is given. In the caching context, the search key is derived from the reference address, and the return data is the data stored at that address. Caches must be very very fast, which means that the search process has to be abbreviated. This is accomplished by dedicating comparison hardware with every location in the associative memory. Presented with a search key, every comparator looks for a match with the key at its location. At most one comparator will see a match and return the data; it is possible that none will. A *fully associative* cache is one where any address can appear anywhere in the cache. This means building the cache to have a unique comparator associated with every address in the cache; doing so is prohibitively expensive. Tricks are played with memory addresses in order to reduce the costs greatly. The idea is to partition the address space into sets. Figure 14.11 illustrates how a 48-bit

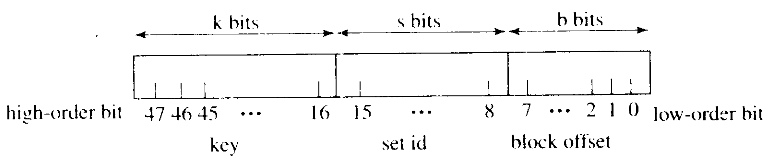


Figure 14.11 48-bit address partitioned for cache.

memory address might be partitioned in key, set id, and block offset. Any given memory address is mapped to the set identified by its set-id address bits. This scheme assigns the first block of 2^b addresses to set 0, the second block of 2^b addresses to set 1, and so on, wrapping around back to set 0 after 2^s blocks have been assigned. Each set is give a small portion of the cache—the set size—typically, 2 or 4 or 8 words. Only those addresses mapped to the same set compete for storage in that space. Only as many comparators are needed as there are words in the set. Given an address, the hardware uses the set-id bits to identify the set number and the key bits to identify the key. The hardware matches the keys of the blocks already in the identified set to comparator inputs and also provides the key of the sought address as input to all the comparators. Comparisons are made in parallel; in the case of a match, the block-offset bits are used to index into the identified block to select the particular address being referenced.

The overall size of this cache is seen to be the total number of sets times the set size. One role of simulation is to work out, for a given cache size, how the space ought to be partitioned into sets. This is largely a cost consideration, for increasing the set size (thereby reducing the number of sets) typically increases the hit ratio. However, if a set size of 4 yields a sufficiently large hit ratio, then there is little point to increasing the set size (and cost).

Least Recently Used (LRU) is the replacement policy most typically used. When a reference is made but is not found in a set, some block in the set is ejected to make room for the one containing the new reference. Under LRU, the block selected for ejection is the one which, among all blocks in the set, was last referenced most distantly in the past.

LRU is one of several replacement policies known as *stack* policies. (See Stone [1990].) These are characterized by the behavior that, for any reference in any reference string, if that reference misses in a cache of size n, then it also misses in every cache of size $m < n$, and that, if it hits in a cache of size m, then it hits in every cache of size $n > m$. Simulations can exploit this fact to compute the miss ratio of many different set sizes, in just one pass of the reference string! Suppose that we do not wish to consider any set size larger than 64. Now we conduct the simulation with set sizes of 64. Every block in the cached set is marked with a priority—namely, the temporal index of the last reference made to it (e.g., the block containing the first reference in the string is marked with 1, the block containing the second reference is marked with a 2 (overwriting the 1, if the same as the previous block), etc.). When a block must be replaced, the one with the smallest index is selected. Imagine that the simulation organizes and maintains the contents of a cached set in LRU order, with the most recently referenced block first in the order. The *stack distance* of a block in this list is its distance from the front; the most recently referenced block has stack distance 1, the block referenced next most recently has stack distance 2, the LRU block has stack distance 64. Presented with a reference, the simulation searches the list of cache blocks for a match. If no match is found, then, by the stack property, no match will be found in any cache of a size smaller than 64, on this reference, for this reference string. If a match is found and the block has stack distance k, then no match will be found in any cache smaller than size k, and a match will always be found in a cache of size larger than k. Rather than record a hit or miss, one increments the k^{th} element of a 64-element array that records the number of matches at each LRU level. To find out how many hits occurred in a cache of size n, one sums up the counts of the first n elements of the array. Thus, with a little arithmetic at the end of the run, one can count (for each set cache) the number of hits for every set of every size between 1 and 64.

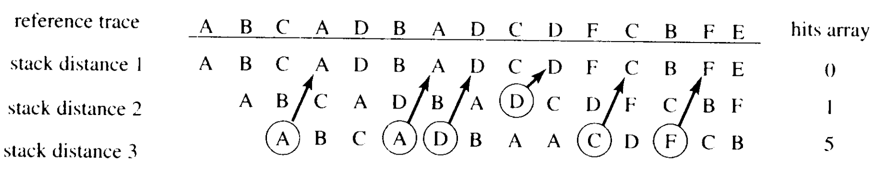

reference trace
A B C A D B A D C D F C B F E
hits array
stack distance 1
A B C A D B A D C D F C B F E
0
stack distance 2
A B C A D B A D C D F C B F
1
stack distance 3
A B C A D B A A C D F C B
5

Figure 14.12 LRU stack evolution

Figure 14.12 illustrates the evolution of an LRU list in response to a reference string. Under each reference (given as a alphabetic symbol rather than actual memory address) is the state of the LRU stack *after* the reference is processed. The horizontal direction from left to right symbolizes the trace, read from left to right. A hit is illustrated by a circle, with an arrow showing the migration of the symbol to the top of the heap. The "hits" array counts the number of hits found at each stack distance. Thus we see that a cache of size 1 will have the hit ratio 0/15, a cache of size 1 will have the hit ratio 1/15, and a cache of size 3 will have the hit ratio 6/15.

In the context of a set-associative cache simulation, each set must be managed separately, as shown in the figure. In one pass, one can get hit ratios for varying set sizes, but it is important to note that each change in set size corresponds to a change in the overall size of the entire cache. This technique alone does not let us in one pass discover the hit ratios for all the different ways one might partition a cache of a given capacity (e.g., 256 sets with set size 1 versus 128 sets with set size 2 versus 64 sets with set size 4). It actually is possible to evaluate all these possibilities in one pass, but the technique is beyond the scope of this discussion.

14.7 SUMMARY

This chapter looked at the broad area of simulating computer systems. It emphasized that computer-system simulations are performed at a number of levels of abstraction. Inevitably, it discussed a good deal of computer science along with the simulation aspects, for in computer-systems simulation the two are inseparable.

The chapter outlined fundamental implementation issues behind computer-system simulators—principally, how process orientation is implemented and how object-oriented concepts such as inheritance are fruitfully employed. Next it considered model input, ranging from stochastically generated traffic, to stochastically generated memory-referencing patterns, to measured traces and direct-execution techniques. The chapter was brought to a conclusion by looking at examples of simulation at different levels of abstraction: a WWW-site server system, an instruction-level CPU simulation, and simulation of set-associative memory systems.

The main point is that computer-system simulators are tailored to the tasks at hand. Appropriate levels of abstraction need to be chosen, as must appropriate simulation techniques.

REFERENCES

ASHENDEN, P.J. [2001], *The Designer's Guide to VHDL*, 2d ed., Morgan Kaufmann, San Fransisco.

AUSTIN T., E. LARSON, AND D. ERNST [2002], "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, Vol. 35, No. 2, pp. 59–67.

BEDICHECK, R.C. [1995], "Talisman: Fast and Accurate Multicomputer Simulation," *Proceedings of the 1995 ACM SIGMETRICS Conference*, pp. 14–24, Ottawa, ON, May.

BOHRER P., M. ELNOZAHY, A. GHEITH, G. LEFURGY, T. NAKRA, J. PETERSON, R. RAJAMONY, R. ROCKHOLD, H. SHAFI, R. SIMPSON, E. SPEIGHT, K. SUDEEP, E. VAN HENSGERGEN, AND L. ZHANG [2004], "Mambo—A Full System Simulator for the PowerPC Architecture," *Performance Evaluation Review*, Vol. 31, No. 4, 8–12.

CMELIK, B., AND D. KEPPLE [1994], "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proceedings of the 1994 ACM SIGMETRICS Conference*, pp. 128–137, Nashville, TN, May.

COVINGTON, R., S. DWARKADA, J. JUMP, S. MADALA, AND J. SINCLAIR [1991], "Efficient Simulation of Parallel Computer Systems," *International Journal on Computer Simulation*, vol. 1, No. 1, 1991.

COWIE, J., A. OGIELSKI, AND D. NICOL [1999], "Modeling the Global Internet," Vol. 1, No. 1, pp. 42–50.

DICKENS, P., P. HEIDELBERGER, AND D. NICOL [1996], "Parallelized Direct Execution Simulation of Message Passing Programs," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7, No. 10, pp. 1090–1105.

FISCHER, W., AND K. MEIER-HELLSTERN [1993], "The Markov-Modulated Poisson (MMPP) Cookbook," *Performance Evaluation*, Vol. 18, No. 2, pp. 149–171.

FISHWICK, P. [1992], "SIMPACK: Getting Started with Simulation Programming in C and C++," *Proceedings of the 1992 Winter Simulation Conference*, pp. 154–162, Washington, DC.

GRUNWALD, D. [1995], *User's Guide to Awesime-II*, Department of Computer Science, Univ. of Colorado, Boulder, CO.

HENNESSY, J.L., AND D.A. PATTERSON [1997], *Computer Organization and Design, The Hardware/Software Interface*, 2d ed., Morgan Kaufmann Publishers, Inc., Palo Alto, CA.

HUGHES, C., V. PAI, P. RANGANATHAN, AND S. ADVE [2002], "RSIM: Simulating Shared Memory Multiprocessors with ILP Processors," *IEEE Computer*, Vol. 35, No. 2, pp. 40–49.

LEBECK, A., AND D. WOOD [1997], "Active Memory: A New Abstraction for Memory System Simulation," *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 1, pp. 42–77.

LITTLE, M.C., AND D.L. McCue [1994], "Construction and Use of A Simulation Package in C++," *C User's Journal*, Vol. 3, No. 12.

MAGNUSSON, P., M. CHRISTENSSON, J. ESKILSON, D. FORSGREN, G. HALLBER, J. HOGBERG, F. LARSSON, A. MOESTEDT, AND B. WERNER [2002], "SIMICS: A Full System Simulation Platform," *IEEE Computer*, Vol. 35, No. 2, pp. 50–58.

MANO, M. [1993], *Computer Systems Architecture*, 3d ed., Prentice Hall, Englewood Cliffs, NJ.

NUTT, G. [2004], *Operating Systems, A Modern Prespective*, 3d ed., Addison-Wesley, Reading MA.

SCHWETMAN, H. [1986], "CSIM: AC-Based, Process-Oriented Simulation Language," *Proceedings of the* 1986 *Winter Simulation Conference*, pp. 387–396.

SCHWETMAN, H.D. [2001], "CSIM 19: A Powerful Tool for Building Systems Models," *Proceedings of the* 2001 *Winter Simulation Conference*, pp. 250–255.

STONE, H. [1990], *High Performance Computer Architecture*, Addison--Wesley, Reading MA.

WITCHEL, E., AND M. ROSENBLUM [1996], "EMBRA: Fast and Flexible Machine Simulation," *Proceedings of the 1996 ACM SIGMETRICS Conference*, pp. 68–79, Philedelphia, PA, May.

EXERCISES

1. Sketch the logic of an event-oriented model of an M/M/1 queue. Estimate the number of events executed when processing the arrival of 5000 jobs. How many context switches on average does a process-oriented implementation of this queue incur if patterned after the SSF implementation of the single-server queue in Chapter 4?

2. For each of the systems listed, sketch the logic of a process-oriented model and of an event-oriented model. For both approaches, develop and simulate the model in any language:
    - a central-server queueing model: when a job leaves the CPU queue, it joins the I/O queue with shortest length.
    - a queueing model of a database system, that implements fork join: a job receives service in two parts. When it first enters the server it spends a small amount of simulation time generating a random number of requests to disks. It then suspends (freeing the server) until such time as *all* the requests it made have finished, and then enqueues for its second phase of service, where it spends a larger amount of simulation time, before finally exiting. Disks may serve requests from various jobs concurrently, but serve them using FCFS ordering. Your model should report on the

statistics of a job in service—how long (on average) it waited for phase 1, how long it waits on average for its I/O requests to complete, and how long it waits on average for service after its I/O requests complete.

3. Consider a three-state (OFF, ON, and BURSTY) Markov Modulated Process with the following characteristics
   (a) The MMP is in ON state for 90% of the time on average.
   (b) The MMP is in BURSTY state for 5% of the time on average.
   (c) OFF to ON transitions probability is 0.8 and OFF to BURSTY is 0.05.
   (d) ON to OFF transition probability is 0.9 and ON to BURSTY is 1.
   (e) BURSTY to ON transition probability is 0.5 and BURSTY to OFF is 0.5.

   If the time spent in OFF state is exponential with a mean of 0.3, determine exponential mean values of time spent in ON and BURSTY states by means of simulation.

4. Recall the pseudo-code for generating reference traces (Figure 14.7). Write routines `new_wrkset`, `from_wrkset`, and `not_from_wrkset` to model the following types of programs:
   (a) a scientific program with a large working set during initialization, a small working set for the bulk of the computation, and a different working set to complete the computation. (You will need to modify the control code in the figure slightly to force phase transitions in desired places):
   (b) a program whose working set always contains a core set of pages present in every phase, with the rest of the pages clustered elsewhere in the address space.

5. Consider computer network with three printers (a, b, and c). The type of printer (a or b or c) is selected by the user and some users are high-priority users. Simulate the model using any simulator or language.

6. Using any simulator or language you like, model the router-to-Web-server logic of the system described in section 14.1. Pay special attention to the load-balancing mechanism that the router employs.

7. Using any simulator or language you like, model the interaction between application server and data server described in section 14.4. Pay special attention to the logic of requesting multiple data services and of waiting until all are completed until advancing to the next burst.

8. Consider the following language for describing CPU instructions:

   ```
   op r1 r2
   ```

   The preceding expressions describe an operation, where

   ```
   op=1  means add, op=2 means subtract. Each require 1 cycle.
   op=3  means mult. r1 receives the result r1 op r2. A multiplication requires 2
         cycles.
   op=4  means a load from memory, into r1, using the value in r2 as the memory
         address. Every 10th load requires 4 cycles, the remaining loads require 1.
   op=5  means a store to memory, storing the data found in r1, using the value in
         r2 as the memory address. Each store requires 1 cycle.
   ```

   Write a CPU simulation along the lines of that described in 14.5 that accepts a stream of instructions in the format just described. Your simulator should use a logical-to-physical register mapping, use the timing information previously sketched, and use stall instructions as described in the example.

9. Integrate the trace generator created in Problem 4 with the one-pass simulator written in the previous problem, in effect creating a pseudo "direct-execution simulator."

10. Analyze the log of WWW requests to your site's server, produce a stochastic model of the request stream, and simulate it.

15

Simulation of Computer Networks

15.1 INTRODUCTION

Computers and the networks that connect them have become part of modern working life. In this chapter, we illustrate by example some of the ways that discrete-event simulation is used to understand network systems, the software that controls them, and the traffic that they carry.

Like computer systems, network systems exhibit complexity at multiple layers. Networked systems are designed (with varying degrees of fidelity) in accordance with the so-called Open System Interconnection (OSI) Stack Mode (Zimmerman, 1980). The fundamental idea is that each layer provides certain services and guarantees to the layer above it. An application or protocol at a particular layer communicates only with protocols directly above and below it in the stack, implementing communication with a corresponding application or protocol at the same stack layer in a different device. Simulation is used to study behavior at all these layers, although not generally all in the same model. Different layers encapsulate different levels of communication abstraction.

The **Physical Layer** is concerned with the communication of a raw bit-stream, over a physical medium. The specification of a physical layer has to address all the physical aspects of the communication: voltage or radio signal strength, standards for connecting a physical device to the medium, and so on. Models of this layer describe physics.

The **Data Link Layer** implements the communication of so-called *data-frames*, which contain a limited chunk of data and some addressing information. Protocols at the Data Link Layer interact with the physical layer to send and receive frames, but also provide the service of "error-free" communication to the layer above it. Protocols at the Data Link Layer must therefore implement error-detection and retransmission when needed. A critical component of avoiding errors is access control, which ensures that at most one device is transmitting at a time on a shared medium. Techniques for access control have significant impact on how long it takes to deliver data and on the overall capacity of the network to move data. Simulation plays an important role in understanding tradeoffs between access-control techniques; in this chapter, we will look at some protocols and the characteristics that simulation reveals.

The **Network Layer** is responsible for all aspects of delivering data frames across subnetworks. A given frame may cross multiple physical mediums en-route to its final destination; the Network Layer is responsible for logical addresses across subnets, for routing across subnets, for flow control, and so on. The success of the Internet is due in no small part to widespread adaption of the *Internet Protocol*, more commonly known as IP (Comer, 2000). IP specifies a global addressing scheme that allows communication between devices across the globe. The specification of IP packets includes fields that describe the type of data being carried in the packet, the size of the packet, the protocol suitable for interpreting the packet, source/destination addressing information, and more. The Network Layer provides error-free end-to-end delivery of packets to the layer above it. Simulation is frequently used to study algorithms that manage devices (routers) that implement the Network Layer.

The **Transport Layer** accepts a message from the layer above, segments it into packets that are passed to the Network Layer for transmission, and provides the assurance that received packets are delivered to the layer above in the order in which they appear in the original message, error free, without loss, and without duplication. Thus, the Transport Layer protocol in the sending device coordinates with the Transport Layer protocol in the receiving device in such a way that the receiving device can infer packet-order information. Variants of the Transmission Control Protocol (TCP) are most commonly used at this layer of the stack (Comer, 2000). Dealing with packet loss is the responsibility of the transport layer. Packet loss is distinct from error-free transmission—a packet could be transmitted to a routing device without error, only to find that device does not have the buffering capacity to store it; the packet is received without error, but is deliberately dropped. Transport layer protocols need to detect and react to packet loss, because they're responsible for replacing the packets that are dropped. One of the ways they do this is to apply flow-control algorithms that simultaneously try to utilize the available bandwidth fully, yet avoid the loss of packets. Simulation has historically played a critical role in studying the behavior of different transport protocols, and in this chapter we will examine simulation of TCP.

The first four OSI layers are well defined and separated in actual implementation. The remaining three have not emerged so strongly in practice. Officially the **Session Layer** is responsible for the creation, maintenance, and termination of a "session" abstraction, a session being a prolonged period of interaction between two entities. Above this one finds the **Presentation Layer**, whose specification includes conversion between data formats. An increasingly important conversion function is encryption/decryption. Finally, the **Application Layer** serves as the interface between users and network services. Services typically associated with the Application Layer include email, network management tools, remote printer access, and sharing of other computational resources.

Any simulation of networking must include models of data traffic, and so we begin the discussion there. At the time of this writing, the field of traffic modeling is very active, and we bring to the discussion key elements of an exciting area of current work.

Devices with traffic they wish to transmit must somehow gain access to the networks that carry traffic. Our second area of discussion then considers the problem of how devices coordinate to use the network medium, sometimes called Media Access Control (MAC) protocols. Historically, simulation has played an important role in helping engineers to understand the performance of different MAC protocols.

Finally, we describe the Transport Control Protocol (TCP) and discuss how simulation plays an important role in its study.

15.2 TRAFFIC MODELING

Our discussion of network simulation begins with modeling of the data traffic that the networks carry. We'll consider two levels of detail for this, corresponding to two different levels of abstraction. The first is at the application level: consideration of how commonly used network applications create demand for a network. Such models are appropriate when one's interest is in the details of a relatively small network and the impact that its native applications have on it. The second level of abstraction is of aggregated application flows. This level is appropriate when one's focus is on the Internet's core infrastructure, where the global impact of global traffic needs to be represented.

One of the easiest models of traffic-load generation is that of moving files across the network. Our interest here is not in the mechanics of the protocols that accomplish the movement so much as it is in the model of the traffic load that is offered to the network. Simulation studies that model file transfer typically are focused on the impact that the traffic has on servers holding the files. A given transfer can be characterized by the size of the file and by the rate at which its bytes are presented to the network. We usually also characterize how often a user initiates an ftp transfer. A simple model of a file-transfer request process is as an on–off source, whose *off* period is randomly distributed (e.g., an exponential think time) and whose *on* period is driven by the arrival of a file. The *on* period lasts as long as needed to push or pull a file of the referenced length. File size is sampled from another probability distribution. Measurements suggest that a heavy-tailed distribution is appropriate. This is especially appropriate given the level of music-sharing activity on the Internet.

Another significant source of application traffic is the World Wide Web. Traffic associated with web pages is more complex than individual file transfers and so bears separate treatment. We describe a model expounded upon in (Barford and Crovella [1998]), called Surge. Here we model the delay between successive sessions with an intersession delay distribution. Within a session, a number of different URLs will be accessed, with another delay time between each such access; this is illustrated in Figure 15.1.

The Surge model incorporates a number of important characteristics of files, most importantly, including

- the distribution of file sizes, among all files on a web server,
- the distribution of the file sizes of those files that are actually *requested*,
- temporal locality of file-referenced file.

The first and third characteristics, coupled with a model of referencing pattern, essentially define the second characteristic. Suppose that we've selected the first k files already—call them $f_1, f_2, \ldots, f_k$—and suppose that this set of references is organized in a Least-Recently-Used stack. We select the $(k + 1)^{st}$ file by sampling an integer from a stack-distance distribution. If that sample has value j, the next file selected has position j in the LRU stack (position 1 being the last file referenced). Empirical studies of reference strings of files suggest that a lognormal distribution is appropriate. This distribution places significant weight on small values; hence, it induces temporal locality of reference. When the stack-distance sample is larger than the number of files in the LRU stack, a new file is sampled from the set of files not yet in the reference stream.

This description gives a general, but simplistic idea of the structure of Surge. Its authors pay much attention to issues of identifying distributional parameters that are internally consistent and that produce traffic that can be validated against real traffic. Our goal here is to introduce the fundamental notions behind a model of web traffic.

Models of other interesting and important application types can be found in the literature. We expect that the Internet will increasingly support telephony—"voice over IP (VoIP)" (Black [2001]), and so attendant models should be developed. A sampling of the current literature suggests that a VoIP source be

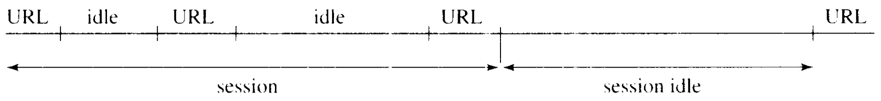


Figure 15.1 Nested on–off periods in Surge WWW traffic generation.

modeled as an on–off process, where both phases have distributions with tails somewhat heavier than exponential (e.g., an appropriate Weibull). Increasingly, the Internet will be used to stream video content. Models for video are more complex, because they must capture a number of facets of video compression, at different time-scales.

All of the application models we've considered describe the traffic workload offered to a network by individual programs. There are contexts in which a modeler needs instead to consider the impact of aggregated application flows on a network device. One *could* create the aggregate stream by piecing together many individual application streams—or one could start with an aggregated model in the first place. We next consider direct models of aggregated offered load.

Classical models of telephone traffic assume that aggregated call arrivals to the telephony network follow a Poisson distribution and that call completions likewise are Poisson. The early days of modeling and engineering *data* networks made the same assumption. However, with time, it became clear that this assumption didn't match reality well. In telephony, the increased use of faxes, and then Internet connections, radically transformed the statistical behavior of traffic. Two things emerged as being particularly different: First, data traffic exhibits a burstiness that flies in the face of the exponential's memoryless property. MMP processes described in Chapter 14 can be used to introduce burstiness explicitly into the arrival pattern of packets to a data network. However, studies indicate that the durations of burstiness aren't Markovian, as in the MMP model. Instead, traffic seems to exhibit long-term temporal dependence—correlations in the number of active sessions that extend past what, statistically, can be expected from MPP models.

Researchers noticed that there is tremendous variance in the size of files transferred within a session. It seemed that a heavy-tailed distribution like the Pareto does a good job of capturing this spread. Heavy-tailed distributions have the characteristic that, infrequently, very very large samples emerge. These large samples are large enough relative to their probability to exert a very significant influence on the moments of the distribution; in some cases, the integral defining variance diverges. It was hypothesized then that long-range dependence in session counts was due to the correlations induced by the concurrency of very long-lived sessions.

A model that appears to capture these explanations is the "Poisson Pareto Burst Process" (Zukeman *et al.* [2003]), in which bursts (e.g., sessions) of traffic arrive as a Poisson process. Each session length has duration sampled from a Pareto distribution. Bursts may be concurrent. More formally, let t_i be the arrival time of the ith burst, equal to $t_{i-1} + e_i$ where e_i is sampled from an exponential, and let b_i be the Pareto-sampled duration of that burst, and let $d_i = t_i + b_i$ be the finishing time of the ith burst. The state at t, $X(t)$, is the number of bursts t_j with $t_j \le t \le d_j$.

The Pareto distribution with parameters a and b has the probability distribution function

$$D(x) = 1 - \left(\frac{b}{x}\right)^a$$

for $x \ge b$. The distribution has mean $(ab)/(a-1)$ and variance $ab^2/((a-1)^2(a-2))$. One can sample a Pareto with these parameters, using the inverse transform technique:

$$x = b \times (1.0 - U)^{-1.0/a}$$

In this equation U is a uniformly distributed random variable.

It is instructive to consider how traffic is analyzed for evidence of long-range dependence and whether the style of synthetic traffic generation described here exhibits it. Let $X_1, X_2, \ldots,$ be a stationary time series, whose samples have mean μ and variance σ^2. The autocorrelation function $\rho(k)$ describes how well correlated are samples k apart in the time series:

$$\rho(k) = \frac{E[(X_t - \mu)(X_{t+k} - \mu)]}{\sigma^2}$$

The sample autocorrelation function can be constructed from an actual sample by estimating the expectation in the numerator. Long-range dependence is observed when $\rho(k)$ decays slowly as a function of k. Long-range dependence is more formally defined in terms of the autocorrelation function, if there exists a real number $\alpha \in (0, 1)$, and a constant $\beta > 0$ such that

$$\lim_{k \to \infty} \frac{\rho(k)}{\beta k^{-\alpha}} = 1$$

The denominator of this limit describes how slowly $\rho(k)$ needs to go to zero as k increases. The smaller α is, the slower is the degradation. $H = 1 - \alpha/2$ is known as the *Hurst parameter* for the sequence. Values of H with $0.5 < H < 1.0$ define long-range dependence; the larger H is, the more significant is the long-range dependence.

To see evidence that PPBP does yield long-range dependence, we ran an experiment where the mean burst interarrival time was 1 second and the Pareto parameters were $a = 1.1$ and $b = 10$. We computed the sample autocorrelation function, shown in Figure 15.2. Here we see directly that the autocorrelation decays very slowly. We also used the SELFIS tool (Karagiannis *et al.* [2003]) to estimate the Hurst parameter; all of its estimators indicate strong long-range dependence in the sampled series.

Burstiness is not the only consideration in traffic modeling. Traffic intensity exhibits a strong diurnal characteristic—that is, source intensity varies with the source's time of day; furthermore, weekends and holidays behave differently still. To accommodate time-of-day considerations, one can allow the exponential burst interarrival distribution of the PPBP to have a parameter that is dependent on the time of day.

The PPBP describes the number of active sessions $X(t)$ as a function of time. $X(t)$ may be transformed into packet arrival rates, and hence into packets, by including a packet-rate parameter λ. The process $\lambda X(t)$ thus gives an arrival rate of packets from an aggregated set of sources to a network device that handles such.

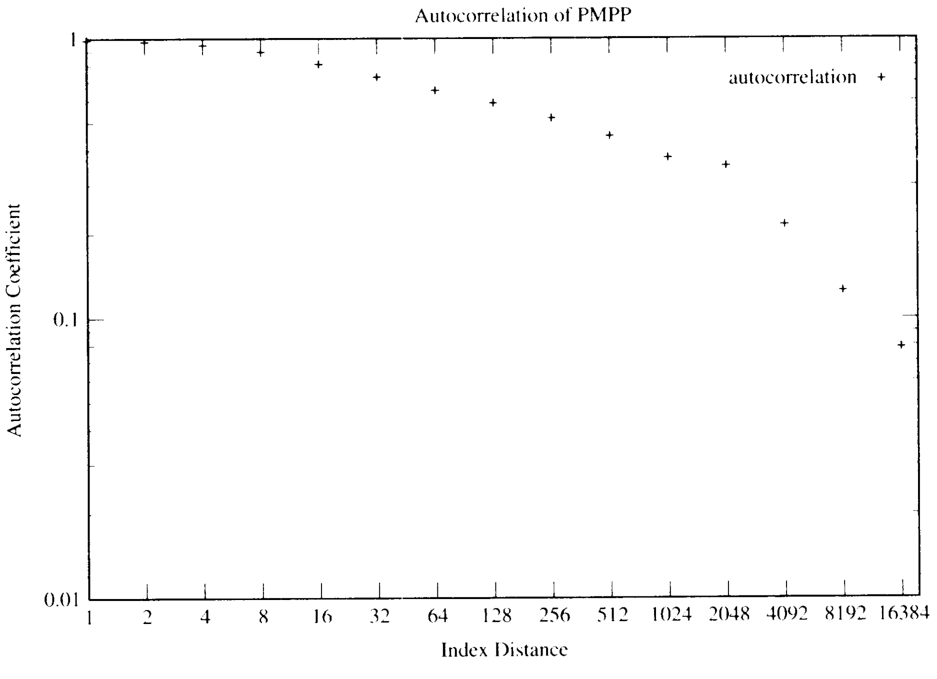


Figure 15.2 Autocorrelation function of aggregated stream of 50 sources.

The main point to be understood about traffic-modeling is that models of aggregated traffic ought to exhibit characteristics of aggregation, whereas application traffic ought to focus on what makes the applications distinct. Next, we look at how traffic acquires a shared medium for carrying traffic.

15.3 MEDIA ACCESS CONTROL

Computers in an office or university environment are usually integrated into a local area network (LAN). Computers access the network through cables (a.k.a. wireline), although an increasing fraction access it through radio (wireless). In either case, when a computer wishes to use the network to transmit some information, it engages in a Media Access Control (MAC) protocol.

Different MAC protocols give traffic different characteristics. Simulation is an extremely important tool for assessing the behavior of a given protocol. A MAC protocol gives traffic specific qualities of latency (average and maximum are usually interesting) and throughput. The behavior of these qualities as a function of "offered load" (traffic intensity) is of critical interest, for some protocols allow throughput to actually *decrease* as the demands on the network go up—a lose–lose situation.

15.3.1 Token-Passing Protocols

One class of MAC protocols is based on the notion of a "token," or permission to transmit. In the "polling protocol" variation, a master controller governs which device on the shared medium may transmit (Kurose and Ross [2002]). The controller selects a network device and sends it the token. If the recipient has "frames" (the basic unit of transmission) buffered up, it sends them, up to a maximum number of frames. The controller listens to the network and detects when the token holder either has selected not to transmit or has finished transmission. The controller then selects another network device and sends it the token. Devices are visited in round-robin fashion.

One drawback of the polling protocol is that the controller is a device with separate functionality from the others. A more homogeneous approach is achieved by using a *token bus* protocol. In this approach a device is programmed to transmit frames (again up to a maximum number) when it receives a token, but is programmed to pass the token directly to a different specified network device after it is finished. There is no controller; the network devices pass the token among themselves, effectively creating a decentralized round-robin polling scheme.

A drawback of both types of token-passing protocol is that a single failure can stop the network in its tracks—in the case of the polling protocol, the network stops if the controller dies; in the case of the token bus, a token passed to a dead device in effect gets lost. In the latter case, one can detect that a device failed to pass the token on and so amend the protocol to deal with like failures.

Token-passing networks are "fair," in the sense that each device is assured its turn within each round. The overhead of access control is the time that the network spends on transmitting the token (rather than data) and the time that the network is idle long enough for a device to ascertain that a transmission has ended or is not going to occur. An important characteristic of token-passing protocols is that the throughput (bits per second of useful traffic) is monotone nondecreasing as a function of the "offered load" (traffic that the network is requested to carry). To illustrate this point, Figure 15.3 plots data from a set of experiments on a modeled 10 Mbits (10 million bits per second) network, with 10 devices, evenly spaced, with a latency delay of 25.6 μsec between the most distant pair. (We use this figure in order to compare this network with one managed by using Ethernet, later.) Five different experiments are displayed on the graph; right now, we are interested only in the one labeled "token bus, Poisson." The experiments assume that the data frame is 1500 bytes long and that the token is 10 bytes long. They assume that, once a device gains the token, it may send at most one frame and then must release the token. This set of data uses a Poisson process to generate frame

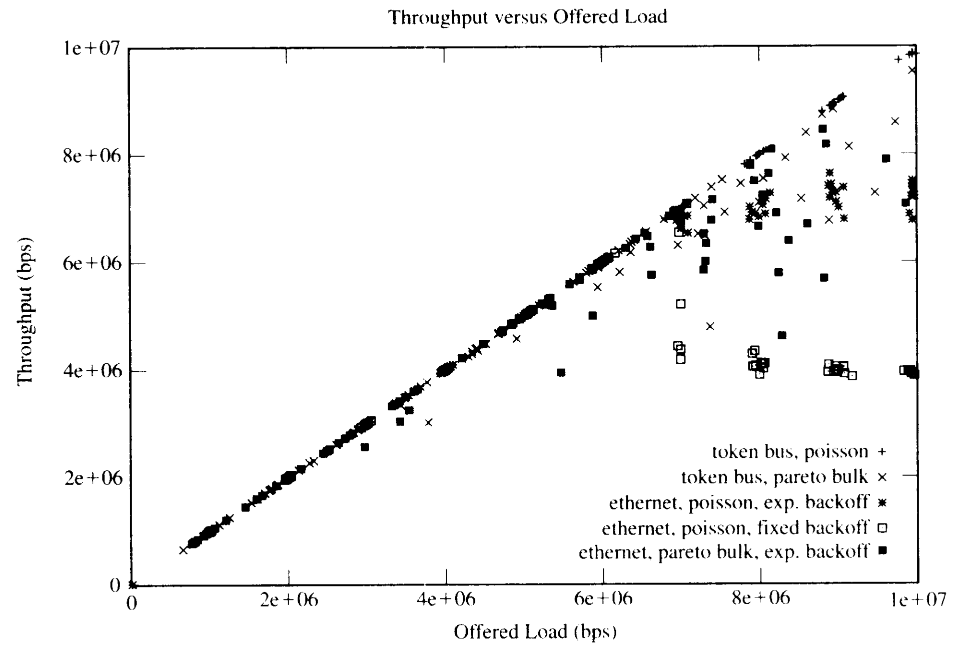


Figure 15.3 Throughput versus Offered Load, for Token Bus and Ethernet MAC protocols, Poisson and Bulk Pareto arrival processes, Exponential and Fixed Backoff (for Ethernet).

arrivals. The x-axis gives the "offered load," measured here as the total sum of bits presented to the network before the simulation end time, divided by the length of the simulation run. The y-axis plots the measured throughput. For each off-time rate, we run 10 independent experiments. For each experiment, we plot the observed pair (offered load, throughput). For the experiment of interest, the throughput increases linearly with the offered load, right up to network saturation. It is interesting to note, though, the impact of a change in the traffic-arrival pattern. We replaced the Poisson arrival process with the arrival process that defines an PPBP, a Poisson bulk arrival process, where the number of frames in each bulk arrival is a truncated Pareto. We use the same Pareto parameters as before ($a = 1.1$ and $b = 10$) and reduced the arrival by a factor of the inverse Pareto mean ($(a-1)/(ab)$) to obtain the same average bit-arrival rate. The set of data points associated with the label "token bus, Pareto bulk" reflect the impact of this change. Throughput grows linearly with offered load until the bus is roughly 60% utilized. For larger loads, we begin to see some deviations from linear. For a point (x, y) off the diagonal, the difference between x and y reflects the volume of unserved frames at the end of the simulation—the frames in queue. This is no surprise; queueing theory tells us that we should expect significant queue lengths when the arrival pattern is highly variant.

Another important aspect is the average time a frame awaits transmission after arrival. Knowledge of queueing theory and the protocol's operation identifies two factors that ought to contribute to growth in the queuing length. One factor is the time required by a token to reach a new frame arrival. As the offered load increases, the amount of work that the token encounters and must serve prior to reaching the new arrival increases linearly. A second factor is from queueing theory; the view from a station is of an M/G/1 queue. In this view, the service time incorporates the time spent waiting for a token to arrive, a mean that increases with the offered load. A job's average time in an M/G/1 queue grows with $1/(1-\rho)$, where $\rho = \lambda/\mu$ is the ratio of arrival rate to service-completion rate. As the offered load grows, ρ increases; this fact explains the second factor of waiting-time growth. As ρ approaches unity, the asymptotic waiting time increases rapidly.

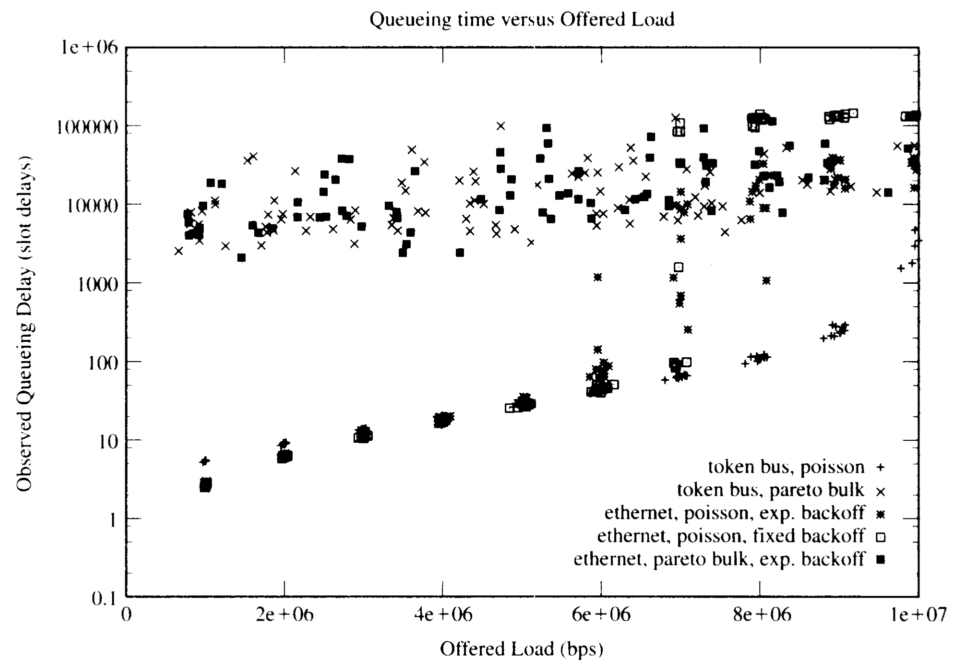


Figure 15.4 Average Queue Delay versus Offered Load: for Token Bus and Ethernet MAC protocols, for Poisson and Bulk Pareto arrival processes, and for Exponential and Fixed Backoff (for Ethernet).

Figure 15.4 confirms this intuition, plotting the average time a frame waits in queue between the time of its arrival and the time at which it begins transmission. Again we execute 10 independent experiments for a given offered load and plot the raw pair (x, q), where x is the average number of bits presented to the network per second in that run and q is the average time a job is enqueued. Units of queueing delay are "slot times", the length of time required for a bit to traverse a cable at the limit of what is permissible for Ethernet (25.6 μsec). The extreme range of queueing delays observed for the five experiment types encourages our use of a log scale on the y-axis. Tracking data from the experiments by using Poisson arrivals, we see stability in the growth pattern, up to the point where the bus is fully saturated. We know to expect extremes there. What is very interesting, though, are the extremely high average queueing delays experienced under the "bulk Pareto" assumptions. If nothing else, these kinds of experiments point out the importance of the traffic model in the analyzing of network behavior.

A straight-forward implementation of a token-bus protocol models devices, the bus, and the explicit and continuous passing of the token among stations. However, this implementation has an undesirable characteristic. Under low traffic load, the model creates a discrete event approximately every 10.84 μsec, the time it takes to transmit a token between adjacent stations. Under low traffic load, the token could completely cycle through the network many times before reaching a point in simulation time when there is a frame available for transmission. Unless the simulation has some particular reason for pushing the token around an otherwise idle network (e.g., if, at each hop, there is a nonzero probability of the token's being lost or corrupted, forcing the protocol to detect and react), there are more efficient ways of executing the simulation, at the cost of incorporating extra logic. We may suppose that each device samples the next future time at which a batch of frames arrives. Before that time, if the device has no frames to transmit, it will make no further demands on the network. When the simulation has reached a time at which no frame is being transmitted and no device has a frame waiting for transmission, we perform a calculation to advance simulation time past an epoch during which the only activity is token passing. Because the time required to circulate the token around the ring is computable, and the next time at which a frame is available at any station is known, we can advance simulation time to the cycle in which the next frame is transmitted and save ourselves the computational effort of getting to that place by pushing a token around.

15.3.2 Ethernet

Token-based access protocols have been popular, but they have drawbacks when it comes to network management. In particular, every time a device is added to or removed from the network, configuration actions must be taken to ensure that a new device gets the token and that a removed device is never again sent the token. The Ethernet access protocol is a solution to this problem (Spurgeon [2000]). A device attached to an Ethernet cable has no specific idea of other devices on that cable; however, when it wants to use the cable, it must coordinate with such other devices. Consider the problem—a device has a frame to send; when can it send it? Ethernet is a decentralized protocol, meaning that there is no controller granting access. A device can "listen" to the Ethernet cable to see whether it is currently in use. If the cable is already in use, the device holds off until the cable is free. However, two or more devices could independently and more or less simultaneously decide to transmit, shortly after which the transmission on the cable is garbled. Both devices can detect this "collision" (e.g., by comparing what they are transmitting on the cable with what they are receiving from the cable). Collision detection and reaction to it is the one of the key components of the Ethernet protocol; it is a so-called Carrier Sense Multiple Access/Collision Detection (CSMA/CD) protocol.

The format of an Ethernet frame is illustrated in Figure 15.5. The 8-byte preamble is a special sequence of bits (alternating 1's and 0's, except for the last bit which is also a '1') that listeners on the cable recognize and use to prepare to examine the next frame field, a 6-byte Destination address that may specify one device, a group of devices, or a broadcast to all listening devices. After scanning the full Destination address, a device listening to the cable knows whether it is an intended recipient. The next 6 bytes identify the sending device; then comes a 2-byte field describing the number of data bytes. The data follow, and the frame is terminated with a 4-byte code used for error detection.

When a device decides to transmit, it begins in the knowledge that it is possible for another device to begin also, not yet having heard the new transmission. Ethernet specifications on network design ensure that any transmission will be heard by another device within $\delta = 25.6$ µsec. This is called a slot time. The worst case is that the device begins to transmit at time t, yet before time $t + \delta$, a device at the other end of the cable decides to transmit and does so just before time $t + \delta$, and another δ time is needed by the first device to detect the collision.

The length of the data portion of an ethernet frame is not specified by the protocol. However, there is a lower bound on the allowable length of the data portion. The frame must be large enough so that it takes longer than 2 slot times to transmit it. This bound ensures that, if a collision does occur, the sending device will be transmitting when the effects of the collision reach it, and hence it can detect the collision. This minimum is 46 bytes of data; furthermore, a frame is not permitted to carry more than 1500 bytes of data.

Some of the complexities of Ethernet exist because of physics. An accurate simulation of Ethernet must therefore pay attention to the delicacies of signal latency. The model used to generate Ethernet performance figures specifically accounts for signal latency. It assumes that the devices are evenly spaced along a cable

size (bytes)	8	6	6	2		4
field	Preamble	Destination MAC adrs	Source MAC adrs	Length	Data	Cyclic Redundancy Check

Figure 15.5 Format of Ethernet Frame.

that requires a full slot time (25.6 μsec) for a signal to traverse. When a device listens to the cable to see whether it is free, the model really answers the question of whether the device can, at that instant, hear any transmission that might have already started. This is a matter of measuring the distance between a sending device and a listening device, computing the signal latency time between them, and working out whether the sender started longer ago than that latency. Likewise, when a device has a frame to send and is listening to the cable to find out when it is idle, its view of the cable state is one that accounts for a certain delay between when a transmission ends and when that end is seen by an observer.

A device with a frame to send listens to the cable and, if it hears nothing begins to transmit. If it successfully transmits the frame without collision and has another frame to send, it waits 2 slot times before making the attempt. If a device wanting to send a frame hears that the cable is in use, it simply waits until the cable is quiet and then begins to transmit. The most interesting part of Ethernet is its approach to collisions. If a device transmitting a frame F detects a collision, it continues to transmit—but jumbled—long enough to ensure that it transmits a full minimum frame's worth of bits. This "jamming" ensures that all devices on the cable detect the collision. Next, it backs off and waits a while before trying to send F again.

The backoff period following a collision has been a topic of some study, one in which simulation has played an important role. If the backoff time is short, there is a chance of not overly increasing the delay time of a frame, but there is also a significant chance of incurring another collision. On the other hand, if the back-off time is large, one reduces the risk of a subsequent collision, but ensures that the delay of the frame in the system will be large. Over time, the following strategy, called "exponential backoff", has become the Ethernet standard. Following the mth collision while attempting to transmit frame F, the device randomly samples an integer k from $[0, 2^m-1]$, and waits $2k$ slot times before making another attempt. If 10 attempts are made without success, the frame is simply dropped. The term "exponential backoff" describes the doubling in length of the mean backoff time on each successive collision. Successive collisions are measures, of a sort, of the level of congestion in the network. A device strives to reduce its contribution to the congestion, and so enable other frames to get through and relieve the congestion.

Simulation is a useful tool to investigate both backoff schemes and other variants of Ethernet one might consider. We did experiments (assuming Poisson arrivals) on exponential backoff and on "fixed" backoff—where, after a collision occurs, the sender chooses $k \in [0, 4]$ slot times to wait, uniformly at random. Figure 15.3 illustrates the effects on throughput. Under exponential backoff, throughput increases linearly with offered load until after about 60% utilization. For greater load, throughput hovers in the 70% of band-width regime, without significant degradation. The story is quite different under fixed backoff. When offered load is 70% of the network bandwidth, the throughput plummets from 60+% and settles in at around 40% of bandwidth—under higher load, the network delivers poorer service. Queueing delays are affected too, as one would expect. Under high load, the delays under fixed backoff are an order of magnitude larger than those under exponential backoff.

A final set of experiments used the same Poisson bulk arrival process, with Pareto-based bulk arrivals, assuming exponential backoff. The results are similar to those for the token bus: large and highly variable queueing delays, and some deviation of throughput from linear at high load. This set of experiments suggests that Ethernet may be more sensitive to the Pareto's high variance than is the token-bus protocol.

15.4 DATA LINK LAYER

A network is far more complicated than the single channel seen by a MAC protocol. A frame might be sent and received many times, by many devices, before it reaches its ultimate destination. Consequently, data traveling at the physical layer contains at least two addresses. One address is a hardware address of the intended endpoint of the current hop. This address (like an Ethernet address) is recognizable by a device's network-interface hardware. The second address is the ultimate destination's network address, typically an

IP address. Different types of devices make up the network. A *hub* is a device that simply copies every bit received on one interface to all its other interfaces. Hubs are useful for connecting separated networks, but have the disadvantage that the connection brings those networks into the same Ethernet collision domain.

A *bridge* makes the same sort of connection, but keeps component subnetworks in different collision domains. For every frame heard on one interface, the bridge takes the destination address and looks up in a table the interface through which that destination can be reached. The bridge has nothing to do if one reaches the destination through the same interface as that through which the frame was observed—the destination will recognize the frame for itself. However, if the destination is reached through a different interface, the bridge takes the responsibility of injecting the frame through that interface, moving it closer to its ultimate goal. In injecting the frame, the bridge acts like a source on that subnetwork, engaging in that subnetwork's MAC protocol. The bridge in effect moves a frame from one collision domain and puts it into another. It can also bridge different subdomain technologies (e.g., different types of Ethernet). Contexts where one would consider simulation study of MAC protocols on one subdomain are the sorts of contexts where one would use simulation and involve models of bridges.

A bridge involves only the physical layer and the data link layer. There is a practical limit on devices retaining the physical addresses of other devices, particularly devices that are in different administrative domains. A *router* is a device that can connect more widely dispersed networks, by making its connections at the Network Layer. A frame coming in to a router on one interface is pushed up to the IP layer, where the IP destination address is extracted; the IP address determines which interface should be used to forward the packet. The *forwarding tables* used to direct traffic flow are the result of complex *routing algorithms*, such as OSPF (Moy [1998]) and BGP (van Beijum [2002]). Simulation is frequently used to study variants and optimizations of these protocols.

We will see that network services commonly used provide users with delivery of data error free and in the order it was sent. These attributes are provided in spite of the real possibility that data will be corrupted in transmission or lost in transmission. A router is one place where a frame might be lost, for, if the router experiences a temporary burst of traffic, all to be routed through a particular interface, buffers holding frames waiting to be forwarded could become exhausted. We think of the traffic flowing through a router as being a set of flows, each flow being defined by the source–destination pair involved. When the arrivals become bursty, and the router's buffer becomes saturated, arrivals that cannot be buffered are deliberately dropped. Most flows actively involved in the burst will lose frames. Under TCP, data loss is the signal that congestion exists, and TCP reacts by significantly decreasing the rate at which it injects traffic into the network. But it takes time to detect this loss—a lot more time than it takes to route frames through the router. One idea that has been studied extensively (by using simulation) is *Random Early Detection* (RED) (Floyd and Jacobson [1993]) queue management. The idea behind RED is to have a router continuously monitor the number of frames enqueued for transmission and, when the average length exceeds a threshold, proactively attempt to throttle back arrival rates before the arrivals overwhelm the buffer and cause all of the flows to suffer. RED visits each frame and, with some probability, either preemptively discards it, or marks a "congestion bit" that is available in the TCP header, but is not much used by most TCP implementations. RED chooses a few flows to suffer for the hoped-for sake of the network as a whole. Complexity abounds in finding effective RED parameters (e.g., threshold queue length, probability of dropping a visited frame) and in assessing tradeoffs and impacts that use of RED could have. Simulation, of course, has played and will play a key role in making these assessments.

15.5 TCP

The Transport Control Protocol (TCP) (Comer [2000]) establishes a connection between two devices, both of which view the communication as a stream of bytes. TCP ensures error-free, in-order delivery of that

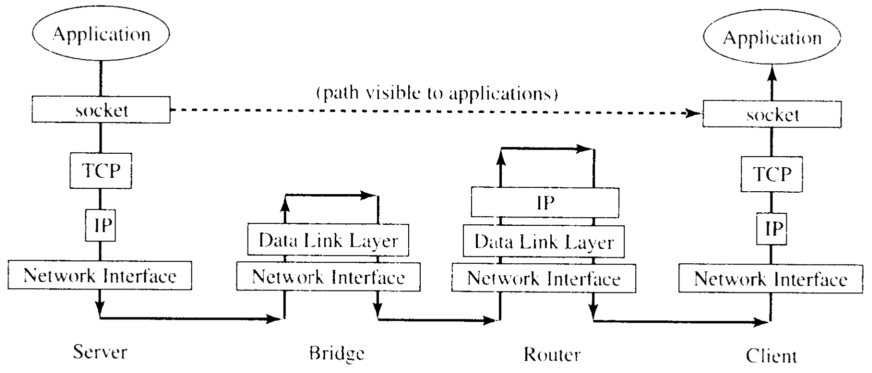


Figure 15.6 Data flow from TCP sender to TCP receiver, passing through network devices.

stream. As we have seen, data frames might be discarded (in response to congestion) somewhere between the sender and receiver; TCP is responsible for recognizing when data loss occurs and for retransmitting data that have gone missing. TCP mechanics are focused on avoiding loss, detecting it, and rapidly responding to it. A number of TCP variants have been proposed and studied; all of these studies use simulation extensively to determine the protocol's behavior under different operating conditions.

Our discussion of TCP serves to illustrate further how different components of networking layers come together. Figure 15.6 illustrates data flow from a server to a client. Two applications intending to communicate establish "sockets" at each side. Sockets are viewed by the applications as buffers into which data could be written and out of which data might be read. Calls to sockets are sometimes blocking calls, in the sense that, if a socket buffer cannot accept more data on a write, or has no data to provide on a read, the calling processes blocks. On the server side, the TCP implementation is responsible for removing data from the socket's buffer and sending it down through the protocol stack to the network. Once on the network, the data pass through different devices. In this figure, we illustrate a bridge (which involves remapping of hardware addresses and does not look at the IP address) and a router (which must decode the IP address to find out the interface through which the data is passed). The client host's IP recognizes that the data ought to go up the stack to TCP, and the client side TCP is responsible for releasing the data to the socket—but only a contiguous stream of data. If the router drops a frame of this flow, the client-side TCP must somehow detect and communicate this absence to the server-side TCP.

TCP segments the data flow into *segments*. Figure 15.7 illustrates the header (in 32-bit words) that is placed around the data. First, note that the only addressing information is "port number" at the source and destination machines—IP is responsible for knowing (and remembering) the identity of the machines involved. From TCP's point of view, there is just a source and a destination. **SeqN** and **AckN** are descriptors of points in the data flow, viewed as a stream of bytes, each numbered. **SeqN** is then the "sequence number" of the first byte in the segment. At the beginning of a connection, a sender and receiver agree upon an initial sequence number (usually random); the **SeqN** value is this initial number plus the byte index within the stream of the first byte carried in the segment. Because the segment size is fixed, the receiver can infer the precise subsequence of the byte stream contained in the segment. The **AckN** field is critical for detecting lost segments. Every time a TCP receiver sends a header about the flow (e.g., in accordance with acknowledgement rules), it puts into the **AckN** field the sequence number of the next byte it needs to receive to maintain a contiguous flow. Since TCP provides a contiguous data stream to the layer above, the value in **AckN** is the initial sequence number plus the index of the next byte it would provide to that layer, if it were available. The linkage of this value with packet loss is subtle. TCP requires a receiver to send an acknowledge for every segment it receives and requires a sender to detect within a certain time limit whether a segment it has sent

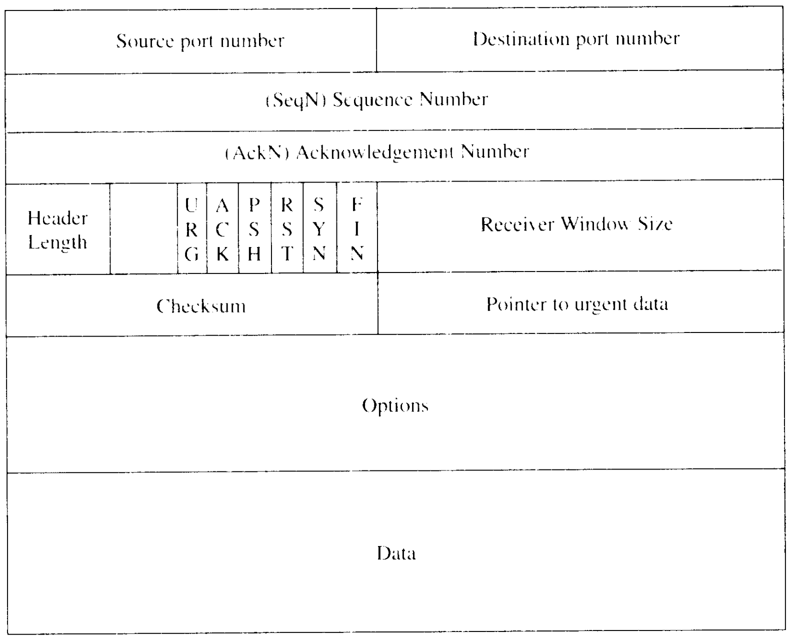


Figure 15.7 TCP header format.

has been acknowledged. Now imagine the effect if 3 segments are sent, and the second one is lost en route. Assume the initial sequence number is 0. The first segment is received, and the receiver sends back an acknowledgement with **AckN** equal to (say) 961 (and the ACK flag set to 1 to indicate that the **AckN** field is valid). The third segment is received, but the receiver notices that the value of **SeqN** is a segment larger than expected—it notices the hole. So it sends back an acknowledgement, but **AckN** in that header is again 961. The second segment sent is not acknowledged, of course, but interestingly, neither is the third. Eventually the TCP sender times out while waiting for these acknowledgements and resends the unacknowledged packets. The only other field in this header, that is critical to our discussion is the Receiver window size, which is included in an Acknowledge to report how many bytes of buffer are currently available to receive data from the sender.

One can visualize TCP as sliding a *send window* over the byte stream. Within the send window are bytes that have been sent, but not yet acknowledged. TCP controls the rate at which it injects segments into the network by maintaining a *congestion window size*, which at any time is the largest the send window is allowed to get. If the send-window size is smaller than the congestion-window size and there are data to send, TCP is free to send it, up until the point where the send window has the same size as the congestion window. When the TCP sender has stopped for this reason, an incoming acknowledgement can reduce the size of the send window (because bytes at the lower end of the window are now acknowledged), and so free more transmission.

TCP tries to find just how much bandwidth it can use for its connection by experimenting with the congestion-window size. When the window is too small, there is bandwidth available but it isn't being used. When the window is too large, the sender contributes to congestion in the network, and the flow could suffer data loss as a result. TCP's philosophy is to grow the congestion window aggressively until there is indication that it has overshot the (unknown) target size, then fall back and advance more slowly. This all is formally described in terms of variables *cwnd* and *ssthresh*. TCP is in *slow start* mode whenever $cwnd < ssthresh$, but in *congestion avoidance* mode whenever *cwnd* > *ssthresh*. Both variables change as TCP executes: *cwnd* grows with acknowledgements a certain way in slow-start, and a different way in congestion-avoidance: *ssthresh* changes when packets are lost. When a TCP connection is first established, *cwnd* is typically set to one segment size and *ssthresh* typically is initialized to a value like 2^{16}. TCP starts in *slow-start* mode, which is distinguished by the characteristic that, for every segment that is acknowledged, *cwnd* grows by a segment's worth of data.

Consider how *cwnd* behaves during slow start by thinking about TCP sending out segments in rounds. In the first round, it sends out one segment, then immediately stalls, because the send-window and congestion-window sizes are equal. When the acknowledgement eventually returns, the sender issues *two* segments as the second round—it replaces the segment that was acknowledged and sends another, because *cwnd* increased by 1. The sender stalls until acknowledgements come in. The *two* acknowledgements for the second round enable the sender to issue *four* segments: half of these due to replacing the ones acknowledged, the other half due to the one-per-acknowledgement increase of *cwnd* rule during slow start. The number of segments issued thus doubles in successive rounds.

Any one of a number of things can halt the doubling of the number of segments sent each round. One is detection of packet loss, the effects of which are to set *ssthresh* to be half the size of the send window, set the send-window size to zero, and set *cwnd* to allow retransmission of one segment (the one in the lost packet). Another way TCP ceases to double the number of segments sent each round is due to the rule that the congestion window may not be increased to exceed certain limits—an internally imposed buffer size at the sender side, or the size of the "receiver window"—the field in ACKs which reports how much space is available for new data. Finally, the doubling effect changes also if *cwnd* grows to exceed *ssthresh*, and so puts TCP into congestion-avoidance mode. Within congestion-avoidance mode, *cwnd* increases, but much more slowly. Intuitively, *cwnd* increases by one segment for every full round that is sent and acknowledged (as opposed to increasing by one segment with every segment that is acknowledged). This is sometimes described as increasing *cwnd* by *1/cwnd* with every acknowledgement.

Simulation is an excellent tool for understanding how TCP works and many of the subleties of its behavior; we now examine simple examples of that behavior. The first topology is that of a server, a client, and a 800 kbps link between them. The server is to send a 300000 byte file to the client. We attach a monitor that emits a tcpdump formatted trace (see `www.tcpdump.crg`) of every TCP packet that passes (in either direction) through the server's network interface. Postprocessing of this trace yields information about how TCP variables of interest behave. In the first situation, we plot the values of **SeqN** in packets sent by the server and the values of **AckN** in packets sent by the receiver in response for the first six rounds, assuming an initial sequence number of 0. This is illustrated in Figure 15.8, where the *Y*-axis is logarithmic in order to illustrate interesting behavior at different scales. The TCP connection is requested by the client at time 192, the first step in TCP's three-way handshake that results in the server sending the first segment at time 192.3 (not actually shown in the graph, to allow higher resolution to later rounds). The **SeqN** in the header of that segment is 1, the index of the first byte in the segment. It takes approximately 100 ms for the segment to reach the client, and another 100 ms for the client's acknowledgment to reach the monitoring point, at time 192.5. (The exact figures are a little different, as they account for the transmission delay caused by the link bandwidth.) The **ACK** bit of that segment is set, and the **AckN** value in the header is 961—the index of the next byte the receiver expects to see. The server's send window now being empty, and *cwnd* having advanced from 1 to 2 by virtue of the received acknowledgement, the server immediately sends *two* segments, one with **SeqN** equal to 961, the next with **SeqN** equal to 961 + 960 = 1921. The graph shows overlapping marks for byte index 961, one from the acknowledgement header, and one from the next segment the server sends. The delay between the server's sending of a segment and the ultimate acknowledgement of that segment is known as the *round-trip time*, or RTT. In this example, the network is as simple as it can be, and the RTT is just the sum of the time to send a segment across the link plus the time to send an acknowledgement back—here, a value very close to 200 ms. At times 192.3 and 192.5, the server stopped sending segments just as soon as

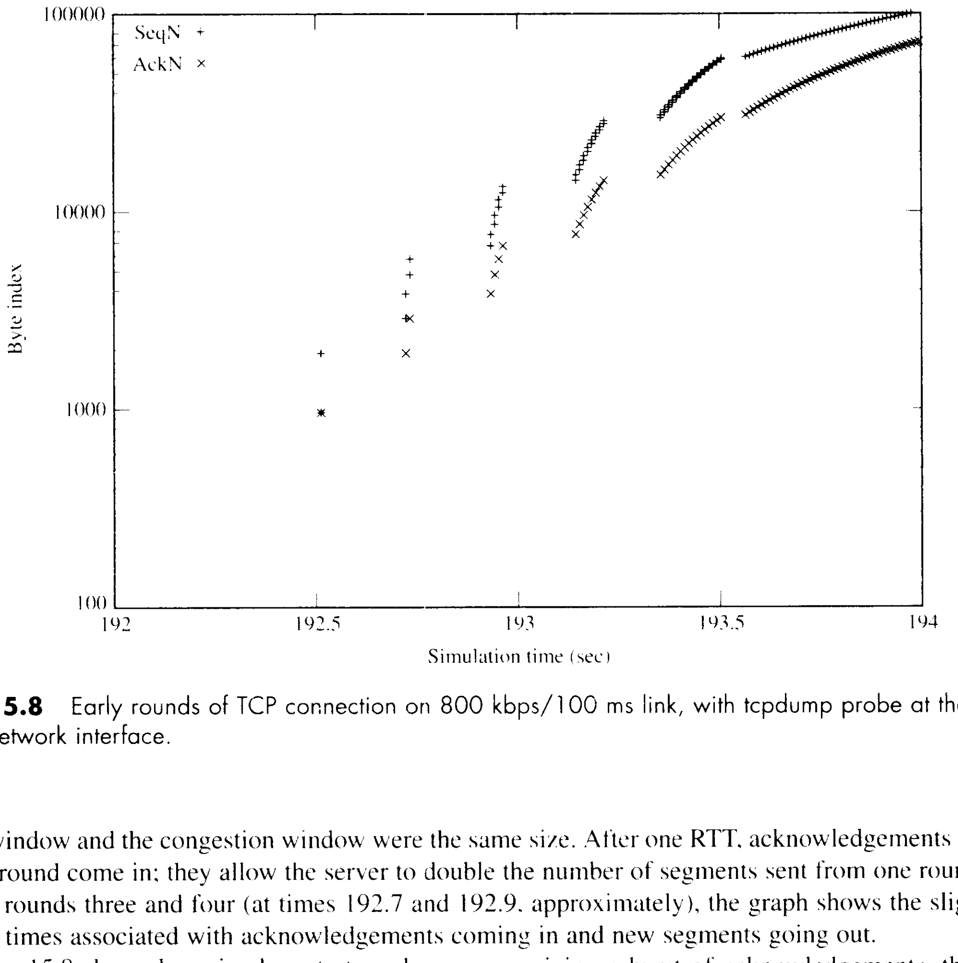


Figure 15.8 Early rounds of TCP connection on 800 kbps/100 ms link, with tcpdump probe at the server's network interface.

its send window and the congestion window were the same size. After one RTT, acknowledgements from the previous round come in; they allow the server to double the number of segments sent from one round to the next. For rounds three and four (at times 192.7 and 192.9, approximately), the graph shows the slight staggering of times associated with acknowledgements coming in and new segments going out.

Figure 15.8 shows how, in slow-start mode, upon receiving a burst of acknowledgements, the server generates a burst of new segments. A moment's reflection shows that, if the acknowledgement for the first segment in that burst is received while the burst is continuing, then the burst will continue ad infinitum. For, at the instant that critical acknowledgement is received, the send window must be smaller than the congestion window, and the send window will not grow after this point, while the congestion window will. We can compute the size of the congestion window at which this phenomenon occurs—it is when the congestion window is large enough that the time needed to transmit that many bytes is precisely the RTT. Back-of-the-envelope calculations indicate that this is 20000 bytes, or just under 21 segments. In these experiments, the receiver window is limited to 32 segments, so this saturation happens before the flow is limited by that buffer. SSFNet initializes *ssthresh* to 65396 bytes, so this saturation point is reached in slow-start, before *cwnd* reaches *ssthresh* and triggers congestion-avoidance mode. Since *cwnd* starts with value 1 and doubles with every round, the server saturates its sends in the middle of the 6th round. This is observed in Figure 15.8, in the round that starts just after time 193.5.

In Figure 15.9, we illustrate this same experiment, along with another that is identical—save that the link latency is 300 ms. A larger epoch of simulation time is illustrated. There is an interesting kink in the **SeqN** data set for the 100 ms network, in the vicinity of **SeqN** = 65*K*. The "slope" of the data set decreases perceptibly. Up to this point, for every acknowledgement received two new segments are transmitted, and they are marked in the tcpdump trace as occurring at the same instant (SSFNet does not ascribe time advance

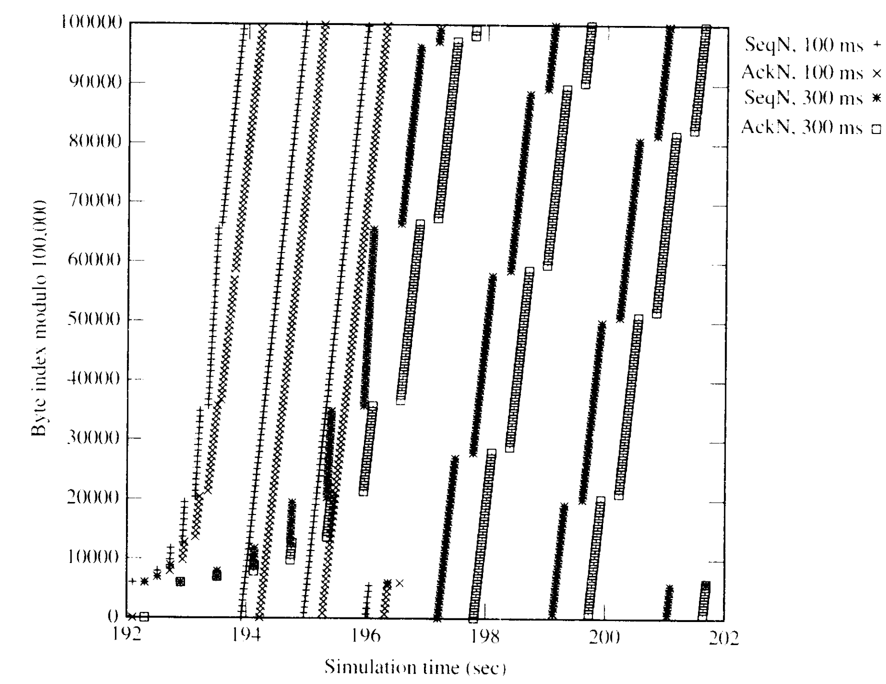


Figure 15.9 TCP connections between server and client: and 800 kbps/100-ms link, and an 800 kbps/300-ms link.

to protocol actions, only to network transmission). At the point of the kink, the value of *cwnd* becomes equal to the receiver window, 32 segments. The sender window becomes limited by the size of the receiver window, rather than by *cwnd*, so, after the kink, there is a one-to-one correspondence between receipt of an acknowledgement and transmission of another segment. Now consider the experiments using a 300 ms latency. As we'd expect, rounds happen approximately every 600 ms. To saturate the link, the sender window has to become three times as large as in the first experiment—almost 64 segments. However, this will never happen, because the send window will be limited by the receiver window, at 32 segments. Indeed, we see that the change in slope of the **SeqN** trace happens at the same byte index as it did with the first experiment. Likewise, we see visually that there's a gap in transmission time between each successive round.

As a final example of how simulation illustrates the behavior of TCP, we consider an experiment designed to induce packet loss. The topology is that of a server, a router, and a client. Again, the server is to send 300000 bytes to the client. Both server and client connect with the router. The link between server and router has 8 Mbps of bandwidth and 5-ms latency. The link between client and router has 800 kbps of bandwidth and 100-ms latency. The router's interface with the client has a 6000-byte buffer. If a packet arrives to that interface and there is insufficient buffer space available, the packet is dropped. From earlier analysis of TCP, we can foresee, in part, what will happen. In the slow-start phase, the server begins to double the number of segments with each successive round. However, it can push packets towards the router 10 times faster than the router can push packets to the client, so a queue will form at the interface. The buffer holds at most 6 packets, so we expect that, in the round where 8 packets are sent, there will be packet loss. Figure 15.10 illustrates this experiment, adding a trace of *cwnd* behavior to that of **SeqN** and **AckN** (once again measured at the server's network interface). The effects of the packet loss are visually distinctive. Around time 193.5, the server begins to receive a sequence of acknowledgements that all carry the same **AckN** value.

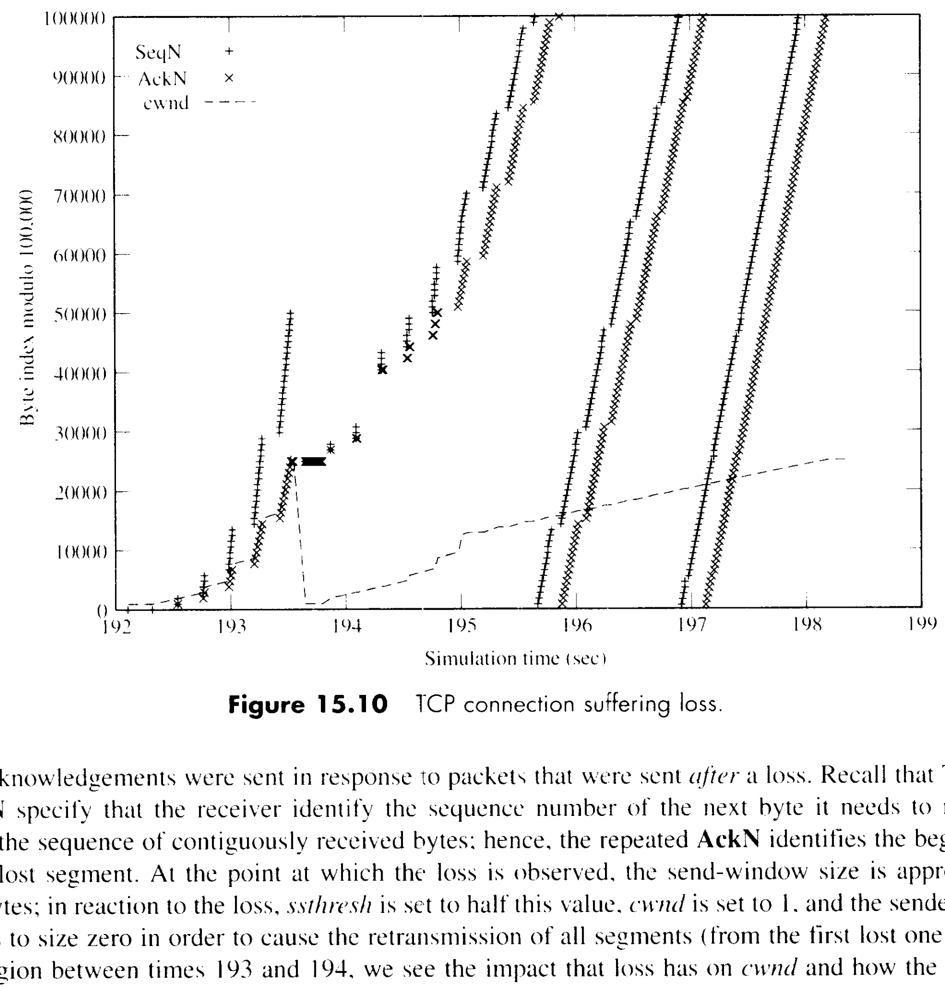


Figure 15.10 TCP connection suffering loss.

These acknowledgements were sent in response to packets that were sent *after* a loss. Recall that TCP rules on **AckN** specify that the receiver identify the sequence number of the next byte it needs to receive to advance the sequence of contiguously received bytes; hence, the repeated **AckN** identifies the beginning of the first lost segment. At the point at which the loss is observed, the send-window size is approximately 25000 bytes; in reaction to the loss, *ssthresh* is set to half this value, *cwnd* is set to 1, and the sender window collapses to size zero in order to cause the retransmission of all segments (from the first lost one forward). In the region between times 193 and 194, we see the impact that loss has on *cwnd* and how the slow-start doubling of *cwnd* with each round begins anew. (Notice the small periods of sharply increased growth at times 194.6, 194.8, and 195.) However, this time, congestion-avoidance mode is entered when *cwnd* reaches *ssthresh*, shortly after time 195; thereafter, it grows more or less linearly with time. This particular transfer ends just before *cwnd* reaches a size that will allow loss once again; had the transfer advanced that far, TCP's treatment of *cwnd* would look very much like the period from 193.8 on.

As these simple examples show, TCP's relatively simple rules create complex behavior. Simulation is an indispensable tool for predicting how TCP will behave in any given context and for understanding that behavior.

15.6 MODEL CONSTRUCTION

SSFNet is a versatile tool for building and analyzing network simulations, used in the previous section to look at how TCP behaves. Suggested homework projects encourage use of SSFNet, and so we describe the general process SSFNet uses in constructing a simulation from an input model. We then illustrate this process, in part, by describing the contents of one input file used in the last subsection. This is not a users' manual for SSFNet; very complete documentation exists at `www.ssfnet.org`. Our aim here to is give a sense of the approach and to encourage readers to investigate further.

15.6.1 Construction

Input to SSFNet is in the form of so-called Domain Modeling Language (DML) files. At the simplest level, a DML file contains just a recursively defined list of attribute–value pairs, where an attribute is a string and a value may be either a string or a list of attribute–value pairs. This structure naturally induces a tree, where interior nodes are attributes (labeled with the attribute string name) and leaves are values of type *string* (rather than of type *list*). To illustrate, consider this DML list:

```
Net [
    frequency 1000000
    host [ id 0
           interface [ id 0 bitrate 800000]
           nhi_route [ dest 1(0) interface 0 ]
    ]
    host [ id 1
           interface [ id 0 bitrate 800000]
           nhi_route [ dest default interface 0 ]
    ]
    link [ attach 0(0) attach 1(0) latency 0.1 ]
]
```

This has some elements of SSFNet DML structure worth noting. Description of a network, elements within the network, and connections between them use a hierarchical naming convention known as the Network-Host-Interface convention, or just NHI. The network is defined in terms of links between interfaces, and each interface has an id number that is unique among all interfaces owned by a common host. That host has an id number that is unique among all hosts in a common net. Each net has an id, unique among all nets contained in the same parent net, and so on. The NHI address 0.1.2(4) refers to an interface named 4, within a host named 2, within a net named 1, within a net named 0. Within a net, a reference such as 2(4) is understood to mean interface 4 associated with the uniquely named host 2 within that understood net. The NHI address of an interface is derived from the nesting described within a DML file. The first interface to appear in the preceding example has NHI address 0(0); the second interface to appear has address 1(0). The `link` attribute in this example specifies two endpoints of the link, in NHI addressing (using the `attach` attribute), and a link latency of 100 ms.

The recursive structure of DML allows it be oxparsed easily and allows one to construct a parse-tree whose interior nodes are attributes and whose leaves are string-valued values. The parse-tree associated with the previous example is illustrated in Figure 15.11. This data structure gives a handy way of methodolically building a model from a DML description. The SSFNet engine recursively traverses the tree and configures core SSFNet objects (such as host). Attributes or values within the tree can be referenced globally by the sequence of attribute labels on nodes from the root to the target. This proves to be useful: one can embed in a DML file a "library" of attribute–value pairs and reference elements of that library.

SSFNet recognizes a variety of attributes, many of which are described in Table 15.1.

15.6.2 Example

Finally, we illustrate some of these ideas by looking at the DML input file for one of our TCP examples. The file is presented in Figure 15.12 (annotated with line numbers for easier reference).

In this particular file, lines 1–8 are comments describing the architecture. Line 10 tells the SSFNet model parser where to find format descriptions of certain constructs; when the parser encounters these constructs in the DML file, it will check against the schema to ensure format correctness. Line 12 starts the

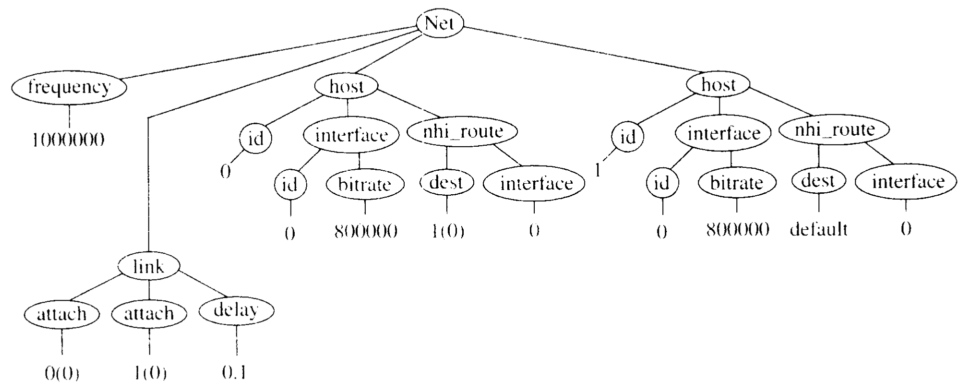


Figure 15.11 Parse tree of simple DML example.

Table 15.1 Common Attributes in SSFNet DML Models

Attribute	*Value*
Net	list describing a network
frequency	number of discrete ticks per simulation second
traffic	list of traffic patterns
pattern	description of traffic pattern, in terms of receiver (client) and server (sender).
servers	list describing a set of servers to which a client might connect—including their NHI identities and port numbers
link	list describing interfaces to be connected, and associated latency
host	list describing a host, and diverse attributes it may have
graph	list of protocols in a host's protocol stack
ProtocolSession	list specifying a protocol
interface	a list describing a connection to the network; attributes include connection bandwidth, and target file for storing monitoring information.
route	description of a forwarding table entry for IP. The *dest* attribute identifies the destination being described; the interface attribute describes which interface packets for that destination should be routed.
dictionary	a list of constants that can be referenced elsewhere within the DML file

overarching list: "Net" followed by a list. Line 14 specifies a clock resolution of 1 microsecond. Lines 15–20 describe the network's traffic, a single pattern that includes host 0 as client. The "servers" attribute gives a list of servers, in this case a single one at NHI address 1(0) (meaning host 1, interface), using port 10.

The "link" attribute at line 24 describes two interfaces to be connected: the one at NHI address 0(0), and the one at NHI address 1(0). The latency across this link is specified to be 0.1 seconds.

A host contains protocols and interfaces to the network. The host beginning at line 28 is given NHI id 1 and contains a "graph" of protocol sessions. Each model of a software component is described as such a session. The order of appearance in the graph is important, descending from higher to lower in the stack. Each protocol session describes its type (e.g., server, client, TCP, IP), and the Java class that describes its behavior. These classes are constructed, by using certain methologies, to be composable; builders of simulation models (in contrast to developers of modeling components these builders use) need not develop new classes,