A MODEL FOR PERFORMANCE ANALYSIS OF COMPUTER SYSTEMS

James W. Kho
California State University, Sacramento

ABSTRACT

Simulation models of computer systems are widely used for system planning and selection assistance. Cost-performance decisions are aided by simulating the operation of various system designs to find the optimum combination of parameters for meeting design specifications. While there may be no theoretical limit to the number of different options one can test, there are very definite limitations in terms of time and cost. To keep the alternatives to a minimum, some reduction method must be employed.

The application of mathematical programming models to computer performance analysis offers a viable solution. This paper begins with the development and use of a simple analytic model using a mathematical programming approach. This is then extended to more complex models of cost-performance analysis. The model equations are defined and algorithmic solutions proved. In a detailed case study, it is shown that the analytic model serves to reduce the number of options to be tested. A simulation model, written in SIMSCRIPT, is incorporated to complete the study.

INTRODUCTION

The evaluation of computer performance is of vital importance in the design of applications and equipment, the selection of computer systems, and the analysis of existing systems. Three general purposes of performance evaluation given in [1] are performance projection, selection evaluation, and performance monitoring. These goals are oriented toward forecasting the impact of changes in the system, designing a new system, or providing data on the actual performance of an existing system.

The introduction of multiprogramming and storage hierarchy brought new levels of complexity to the use and organization of computer systems. With the advent of third generation systems, it has become increasingly difficult to adequately evaluate the factors affecting the performance and efficiency of these systems. Progress in designing and applying information handling systems has outraced progress in evaluating their performance. While simulation models seem to offer the best alternative of existing evaluation tools, two major limitations are time and cost.

An important experimental design problem for many simulation models of computer systems is that of size and factor selection [2]. Each additional replication of the simulation run for a particular design is costly. It is suggested in this paper that the application of mathematical programming models serves well in reducing the number of factors and the alternative levels of each factor in the design.

In the use of analytic models for complex systems, many simplifying assumptions are usually necessary to keep the problem mathematically manageable. The use of expected values may introduce serious distortions in the results. Thus analytic models are often used only to study general phenomena, while finer measurements are made with higher level techniques. A primary motive here is to show that analytic models may be used in solving specific evaluation problems, at least to the extent of narrowing down options.

A serious drawback to solving complex mathematical programming problems is that while formulation may be relatively easy, conventional solutions may often be inapplicable or impractical. The use of conventional methods is circumvented here by developing algorithmic solutions through analysis of the specific nature of the constraints to the problem.

DETERMINING OPTIMUM BUFFER SIZE

This paper begins with a relatively simple design problem for the UNIVAC 1108 operating system [3]. Monitoring of system performance for a particular workload has indicated low CPU productivity. Preliminary analysis points to I/O buffer size as a potential bottleneck. While a simulation model is developed for studying the effect of using different buffer sizes, an analytic model is also constructed to complement it.

In the subsystem relevant to this study, the two main functions performed are (1) execution of programs, including the executive routines, and (2) transfer of data between auxiliary memory and main memory. The subsystem consists of a single central processor, a main storage area, and a peripheral subsystem of I/O and secondary storage devices.

Jobs enter the main memory via I/O devices and reside there while being processed by the CPU. With only a part of the total data file contents of a given job being input, the operation is buffered and parts are processed one at a time. Each job thus undergoes alternating operations of the two functions mentioned: an I/O period, then a compute period, followed by another I/O period, and so on until the job is completed.

Assuming then that there are a number of job segments residing simultaneously in core, each of these segments will be one of the four states: (1) being processed by the CPU, (2) waiting for CPU processing, (3) engaging in I/O activity, or (4) waiting for I/O facilities. The CPU is inactive when all job segments in core are either undergoing I/O activities or waiting for I/O facilities.

The performance criterion used is the index of CPU productivity as defined by

$$\text{CPU productivity} = \frac{\text{Expected CPU busy period}}{\text{Expected CPU busy period} + \text{Expected CPU idle period}}$$

This is the long-run fraction of time the CPU is busy. The success of the multiprogramming system is measured by the value of this index.

It is clear that CPU productivity is improved if for each job segment, the expected compute period may be increased at a rate faster than that of the expected I/O period. The I/O completion time distribution is a function dependent on the buffer size being used for I/O activities and the number of job segments in core simultaneously. This distribution is analyzed in [4] for direct access storage devices. The expected compute period generally increases or decreases as the input buffer size is increased or decreased. With some exceptions*, an increase in the input buffer size will increase the compute period by delaying the demand for the next logical input buffer. A decrease in the input buffer size will hasten the demand for the next logical input buffer.

It must be noted that an increase in input buffer size may cause a corresponding decrease in the number of job segments which can reside in main storage simultaneously. This will significantly affect the expected CPU busy time and hence the index of CPU productivity. This constraint plays heavily in the optimization problem.

---

*For example, user programs accessing libraries and their own private files. These I/O operations use separate buffers and would not be affected by changing input buffer sizes.

The Mathematical Model

Let t_1 and t_2 be the expected compute time and I/O time that each job segment undergoes alternately assuming that buffer size equals one sector capacity, w words. Then the index of CPU productivity γ is

$$\gamma = mt_1/(t_1 + t_2)$$

where m is the mean number of job segments residing in core simultaneously.

If the buffer size is augmented by a multiple j (to j x w words), the expected compute time is expressed as a function of j, $t_1(j)$. It is assumed that $t_1(j)$ is linearly proportional to j and its values may be derived accordingly from t_1. [3] discusses the case where this assumption does not hold. The expected I/O time is a function of j, storage device parameters, and the number of I/O requests queued up in the system, which is in turn dependent upon m. For simplification of notation, let the expected I/O time be $t_2(j,m)$.

The objective function becomes

$$\max \gamma = mt_1(j)/[t_1(j) + t_2(j,m)] \qquad (1)$$

with constraints

$$\gamma \leq 1$$

and m, j = positive integers

Equivalently, (1) may be rewritten as

$$\min \delta = t_1(j) + t_2(j,m) - mt_1(j)$$

with constraints

$$\delta \geq 0$$

and m, j = positive integers

where δ denotes the expected CPU idle time.

Consider the area of main storage used for buffers and for storing the m job segments. Let this have a capacity of C words. Let L be the average length of the job segments. Then

$$C = mL + mjw + r$$

where the surplus variable r satisfies

$$0 \leq r < L + jw$$

This expresses the goal of storing as many tasks in main memory as possible, in order to keep the CPU busy.

The problem may be stated mathematically as

$$\min \delta = (1 - m)t_1(j) + t_2(j,m)$$

subject to

$$\begin{aligned} &\text{(i)} && C = m(L + jw) + r \\ &\text{(ii)} && r < L + jw \\ &\text{(iii)} && m, j \geq 1 \quad \text{integers} \\ &\text{(iv)} && r \geq 0 \quad \text{integers} \\ &\text{(v)} && \delta \geq 0 \end{aligned} \qquad (2)$$

Solution Procedure

The proof of existence of an optimal solution to the problem is given in [3]. The problem is both a nonlinear and an integer programming problem, hence not solvable by conventional method. The procedure described below finds the optimal solution to the minimization problem. It is intended to store the maximum number of tasks, based on core space available, in the main memory while minimizing CPU idle time. In the model, this quantity is determined by the buffer size used. Hence starting with a feasible solution--the minimal buffer size--the objective function is evaluated. This value denotes CPU idle time and equals zero if full utilization is achieved. The buffer size is changed in multiples of the sector capacity. Since the core space in question is large in comparison to buffer and program sizes, increasing buffer size by one sector will either reduce the maximum number of tasks in core by only one or leave it unchanged. In both cases, the change is desirable when CPU idle time is reduced. On the other hand, an increase in objective function value could mean that only a local minimum is found. An analysis of the relation between the objective function and the constraints produces conditions which, when satisfied, imply global optimality. In the algorithm, local minimum points are found in sequence and tested for this optimality.

Algorithm

Step 0. Initialize j at 1 and let

$$m = \left[\frac{C}{jw + L}\right] \qquad (3)$$

where [y] means integer part of y.

Define $\Delta t_1(j) = t_1(j) - t_1(j - 1)$

$$\Delta_j t_2(j,m) = t_2(j,m) - t_2(j - 1,m)$$

and $\Delta_{j,m} t_2(j,m) = t_2(j,m) - t_2(j - 1, m + 1)$

Step 1. Evaluate δ; stop if $\delta \leq 0$, i.e., CPU idle time equals zero.

Step 2. Let j =: j + 1*, $m = \left[\frac{C}{jw + L}\right]$. If m remains unchanged, go to Step 3; otherwise proceed to Step 5.

Step 3. If $(1 - m)\Delta t_1(j) + \Delta_j t_2(j,m) < 0$ (4)

return to Step 1; otherwise go to Step 4.

Step 4. Let $r = c - m((j - 1)w + L)$

$$q = [r/mw] \qquad (5)$$

$$\psi_1 = (1 - m) \sum_{i=0}^{q} \Delta t_1(j + i) + \sum_{i=0}^{q} \Delta_j t_2(j + i,m) \qquad (6)$$

$$\phi_1 = (1 - m)\,\Delta t_1(j + q) + t_1(j + q) + \Delta_{j,m} t_2(j + q, m - 1) \qquad (7)$$

If $\psi_1 + \phi_1 < 0$

go to Step 1; otherwise, stop.

Step 5. If $(1 - m)\Delta t_1(j) - t_1(j - 1) + \Delta_{j,m} t_2(j,m) < 0$ (8)

return to Step 1; otherwise go to Step 6.

---

*This means replace j by its original value plus one.

Step 6. Let $r = C - m(jw + L)$

$$q = [r/mw]$$

$$\psi_2 = (1 - m)\Delta t_1(j) - t_1(j - 1) + \Delta_{j,m} t_2(j,m) \quad (9)$$

$$\phi_2 = (1 - m) \sum_{i=1}^{q} \Delta t_1(j + 1) + \sum_{i=1}^{q} \Delta_j t_2(j + i,m) \quad (10)$$

if $\psi_2 + \phi_2 < 0$

go to Step 1; otherwise, stop.

The proof of the algorithm follows:

Lemma. Given $m_1 = [\frac{C}{jw + L}]$ where [y] means the integer part of y,

and $m_2 = [\frac{C}{(j + q)w + L}] = m_1$,

$$m_3 = [\frac{C}{(j + q + 1)w + L}] = m_1 - 1, \quad (11)$$

then $q = [r/m_1 w]$ where $r = C - m_1(jw + L)$.

Proof: $m_1 = m_2$ implies that

$$[\frac{C}{jw + L}] = [\frac{C}{(j + q)w + L}] = m$$

or $C = m(jw + L) = r_0$ where $0 \leq r_0 < jw + L$

and $C = m((j + q)w + L) + r_q$ where $0 \leq r_q < (j + q)w + L$

thus $r_0 \geq r_0 - r_q$

$= C - m(jw + L) - C + m((j + q)w + L)$

$= mwq$

The maximum value of q where $r_0 \geq mwq$ is (11).

Theorem. The algorithm finds the optimal solution of (1) and (2).

Proof: Due to (i) and (ii) of (2), m is strictly determined for any value of j by (3). Since C >> w, then increasing j by 1 will either not change the value of m or cause it to decrease by 1.

Case (i). j =: j + 1, m unchanged: The objective function δ is affected by an amount of

$$(1 - m)\Delta t_1(j) + \Delta_j t_2(j,m) \quad (12)$$

which is negative if (4) holds. Thus, it would be beneficial to increase the buffer size.

Case (ii). j =: j + 1, m =: m - 1: The objective function δ is unaffected by an amount of

$$(1 - m)\Delta t_1(j) - t_1(j - 1) + \Delta_{j,m} t_2(j,m) \quad (13)$$

If (13) is negative, the buffer size should then be increased.

Suppose (4) does not hold for Case (i) and j may be increased by at most q without affecting m, then letting j =: j + q + 1 will increase δ by ψ_1 given in (6) and decrease it by ϕ_1 in (7). Thus increasing the buffer size would be beneficial and should be made only if the total effect is negative. By the lemma, q is found to be (5).

Suppose (8) does not hold for Case (ii) and j may be increased by at most q without affecting m again, then letting j =: j + q + 1 will increase δ by the amount ψ_2 given in (9) and decrease it by ϕ_2 in (10). The buffer size should be increased only if the total effect decreases δ. Again by the lemma, q is determined by (5).

When at any time (4) and (8) both do not hold, then the algorithm stops and the optimal solution is found. It is noted that (12) and (13) are increasing functions as j increases and m decreases. Thus once either (4) or (8) does not hold, they will remain that way.

Generalization

When several independent I/O mechanism are used, e.g., two drums on separate channels, the model may be modified easily. The objective function may be rewritten as

$$\delta = (1 - m)t_1(j) = \sum_{i=2}^{k} \phi_i t_i(m,j)$$

where each t_i gives the expected completion time of a particular I/O mechanism and ϕ_i its corresponding probability of being utilized. ϕ_i is easily represented by

$$\phi_i = \frac{\text{Expected number of I/O requests for mechanism } i}{\text{Expected total number of I/O requests}}$$

MEMORY HIERARCHY OPTIMIZATION

The optimization of data buffer size is extended here to that of memory hierarchies. The selection involves the types and sizes of memory storage devices such that response time is minimized subject to a cost constraint, or cost is minimized provided that system requirements are met. The models assume an a-priori knowledge of the processing requirements of the system.

Assume that the programs and data may be partitioned into information blocks and characterized by (i) frequency of use, f_i, and (ii) block size, s_i (given in some basic record size, say sector.) Assuming further that program activities are measured from past experience or known by analysis, f_i may be the mean of a range of frequencies. Let

$$F_i = f_i / \Sigma f_k$$

then $\Sigma F_i = 1$

Let n_i be the number of blocks having the same size and frequency of access.

Let the range of storage devices where selection is to be made be characterized by (i) cost per unit, c_j, (ii) capacity, m_j (given in the basic record size), and (iii) speed. Let the response time, T_{ij}, for accessing a block of information on a particular device be a function of both the speed of the device and size of the information block. Thus

$$T_{ij} = t_{1j} + s_i t_{2j}$$

where t_{1j} is the access time and $s_i t_{2j}$ is the transfer time of an information block of size s_i. For direct access devices, t_{1j} stands for, in addition to queue time, rotational delay in the case of drums, and for seek time plus mechanical delay in the case of disks. $t_{1j} = 0$ for random access. For sequential files, I/O response time may be measured by means of the file activity ratio [5].

Model to Minimize Response Time

In this model, the objective function is to minimize the average response time to the information blocks. Let P_{ij} be the fraction of the n_i blocks of information type i to be stored in memory type j. Thus

$$0 \leq P_{ij} \leq 1 \qquad \text{all } i, j$$

and $\sum_j P_{ij} = 1$

The problem is formulated in two ways: with and without the assumption that the memory size of a storage device is divisible. In the first formulation, the problem reduces to a linear programming problem and in the second, it is a mixed integer problem.

Let the cost of storing an information block of size s_i in memory type j be C_{ij}. Thus $C_{ij} = c_j/m_j \times s_i$. Let $V_{ij} = n_i F_i T_{ij}$. Then the objective function is to minimize

$$\gamma = \sum_{ij}\sum V_{ij}P_{ij}$$

The cost constraint is easily seen as

$$\sum_{ij}\sum W_{ij}P_{ij} \leq C_{max}$$

where $W_{ij} = n_i C_{ij}$ and the total cost cannot exceed C_{max}.

An absolute minimum response time requirement may be added by stipulating that all information blocks of type i must be accessed in no less than time a_i, i.e., $T_{ij} \leq a_i$ for all j, where $P_{ij} > 0$. Thus

$$P_{ij}(a_i - T_{ij}) \geq 0$$

On the other hand, an average minimum response time requirement, r_i, may be added by

$$\sum_j P_{ij}T_{ij} \leq r_i$$

When the response time to certain programs or data is more important than others, a weight factor, w_i, may be associated with each type of information block, where $0 \leq w_i \leq 1$. w_i = denotes no importance whatsoever while $w_i = 1$ denotes the utmost importance. V_{ij} may then be modified to be

$$V_{ij} = w_i n_i F_i T_{ij}$$

The optimization problem is then

$$\min \gamma = \sum_{ij}\sum V_{ij}P_{ij}$$

Subject to

distribution	$\sum_j P_{ij} = 1$	all i
	$P_{ij} \geq 0$	all i, j
cost	$\sum_{ij}\sum W_{ij}P_{ij} \leq C_{max}$	
response time	$(a_i - T_{ij})P_{ij} \geq 0$	all i, j
or	$\sum_j T_{ij}P_{ij} \leq r$	all i

Since memory usually consists of indivisible modules, let u_j be the integral number of modules of storage device type j to be in the hierarchy. Then the cost constraint becomes

$$\sum_j c_j u_j \leq C_{max}$$

and an additional capacity constraint is needed, i.e.

$$\sum_i n_i s_i P_{ij} \leq m_j u_j \qquad \text{all } j$$

The optimization problem becomes

$$\min \gamma = \sum_{ij}\sum V_{ij}P_{ij}$$

Subject to

distribution	$\sum_j P_{ij} = 1$	all i
	$P_{ij} \geq 0$	all i, j
cost	$\sum_j c_j u_j \leq C_{max}$	
response time	$(a_i - T_{ij})P_{ij} \geq 0$	all i, j
or	$\sum_i T_{ij}P_{ij} \leq r_i$	all i
capacity	$\sum_i n_i s_i P_{ij} \leq m_j u_j$	all j
modularity	$u_j \geq 0$	integers, all j

Model to Minimize Cost

The objective function here is to minimize the total cost of the memory hierarchy $\sum_{ij}\sum W_{ij}P_{ij}$ with the distribution and response time constraints. Thus

$$\min \alpha = \sum_{ij}\sum W_{ij}P_{ij}$$

subject to

distribution	$\sum_j P_{ij} = 1$	all i
	$P_{ij} \geq 0$	all i, j
response time	$(a_i - T_{ij})P_{ij} \geq 0$	all i, j
or	$\sum_j T_{ij}P_{ij} \leq r_i$	all i

Adding the modularity constraint, the objective cost function becomes $\sum_j u_j c_j$ subject to capacity, response time, and distribution constraints. Thus

$$\min \alpha = \sum_j u_j c_j$$

Subject to

distribution	$\sum_j P_{ij} = 1$	all i
	$P_{ij} \geq 0$	all i, j
capacity	$\sum_i s_i n_i P_{ij} \leq m_j u_j$	all j
response time	$(a_i - T_{ij})P_{ij} \geq 0$	all i, j
or	$\sum_j T_{ij}P_{ij} \leq r_i$	all i
modularity	$u_j \geq 0$ integers	all j

Reconfiguration Models

Let $j = 1,2,\ldots,p$ be the p types of storage devices in an existing memory hierarchy. Let c_j, $1 \le j \le p$, be their resale values and c_j, $j > p$ be the cost of acquiring a new module of memory type j. Let $\bar{u}_j$, $1 \le j \le p$, be the number of modules of memory type j in the existing memory hierarchy. Note that for each type of memory device in the present hierarchy where the resale price and the cost of acquiring a new module is different, it is represented once in $1 \le j \le p$ and another time in $j > p$.

It is generally true that the resale price of a memory module is lower than that of acquiring a new unit of the same type. For that reason, the problem may be formulated as if the entire existing memory hierarchy is sold with the knowledge that part or all of it may be bought back at the resale price. Naturally, the restriction

$$u_j \le \bar{u}_j \qquad j = 1,2,\ldots,p$$

must be imposed to prevent "over rebuying".

The previous models are then easily modified to take into account the possibility of reconfigurating. Let the resale price of the entire existing memory hierarchy be C_R where $C_R = \sum_{j=1}^{p} \bar{u}_j c_j$. Then the minimization of response time model becomes

$$\min \gamma = \sum_i \sum_j V_i p_{ij}$$

Subject to

distribution	$\sum_j p_{ij} = 1$	all i
	$P_{ij} \ge 0$	all i, j
cost	$\sum_j c_j u_j \le C_{max} + C_r$	
modularity	$u_j \ge 0$	integers, all j
capacity	$\sum_i n_i s_i p_{ij} \le m_j u_j$	all j
response time	$(a_i - T_{ij})P_{ij} \ge 0$	all i, j
or	$\sum_j T_{ij} P_{ij} \le r_i$	all i
"rebuying:	$u_j \le \bar{u}_j$	$1 \le j \le p$

Likewise, the minimization of cost model becomes

$$\min \alpha = \sum_j c_j u_j - C_r$$

Subject to

distribution	$\sum_j P_{ij} = 1$	all i
	$P_{ij} \ge 0$	all i, j
capacity	$\sum_i s_i n_i P_{ij} \le m_j u_j$	all j
response time	$(a_i - T_{ij})P_{ij} \ge 0$	all i, j
or	$\sum T_{ij} P_{ij} \le r_i$	all i
modularity	$u_j \ge 0$	integers, all j
"rebuying"	$u_j < \bar{u}_j$	$1 \le j \le p$

Solution Procedure

Without the modularity constraint, the models are constructed as linear programming problems, and with the added constraint, as mixed integer problems. Both may be solved by using conventional methods. [4] provides a set of algorithmic procedures that result in considerable computational time savings over conventional methods.

This is done using a "knapsack problem" approach [6]. Starting from a basic feasible solution where only a limited number of storage device types are considered, the cost limitation is increased in increments. The corresponding minimum average access time is then readily obtained from the previous one. The same is done as the number of memory types being considered is also increased. The optimal solution is found when the maximum cost equals that of the cost constraint and all types of storage devices left after the reduction have been considered. Satisfaction of other constraints, such as the distribution and response time constraints, is maintained throughout the procedure.

In minimizing the average access time subject to a maximum cost constraint, it would be quite beneficial to know if performance could be improved considerably if the cost limitation is raised to a certain level. On the other hand, it might also be the case that performance will not be reduced by much while the cost limitation is lowered. A brute-force method to obtain this information would be to solve the problem many times with different cost constraints. Then plotting the maximum performance versus the corresponding cost would give a curve where the slope indicates the cost-performance ratio.

Undoubtedly, the cost-time curve is quite useful considering that maximization of the cost-performance ratio is generally one of the goals. Unfortunately, the brute-force method is too time consuming and not realistic. Due to the "knapsack problem" approach used in the solution procedure, the table provides the maximum performance for each predetermined incremental cost. Thus the cost-time curve and cost-performance ratios may be obtained directly from the table. In addition, the table also gives this same information where subsets of the storage devices are considered for the optimal memory hierarchy.

A CASE ANALYSIS

Experimental work is done on the data buffer size optimization problem mentioned earlier. The statistics for jobs run at the University of Wisconsin Academic Computing Center in five randomly chosen days are compiled and used for analysis. Core utilization by user programs varies from 1 block of 512 words to 128 blocks. Distributions of the number of jobs versus compute time for the five sample days' runs are gathered. Values obtained for other parameters include number of jobs, means number of programs per job, mean compute time, mean I/O time, number of I/O requests, average request time, number of queue requests, and average queue time. The number of I/O requests is further broken down into the requests for each I/O mechanism which include the FH drums, FASTRAND II movable head drum, and tapes.

Simulation Model

A simulation model is implemented in the SIMSCRIPT 1.5 language. Briefly, the simulation model has the following features. The system consists of two main "first-in, first-out" queues--one for the CPU and another for I/O. At any instant, each task in core is in one of the two queues; it is either (1) executing or enqueued for execution, or (2) waiting for I/O facilities or completion of an I/O operation.

An overloaded operation is assumed in the sense that there is always a task ready to enter the system. A record of the tasks is kept during the entire duration of simulated time, beginning with the scheduling of the first I/O request. Once a job is brought into core, it undergoes a series of I/O and compute periods alternately, remaining in main memory until its completion.

In the model, direct representation of physical devices occurs in the consideration of the drum rotational delay and positioning of read/write heads. A model parameter specifies the core storage. The loading of jobs into core and allocation of their buffer areas are also modeled.

While a change in the distribution of the workload description may alter the system performance, a number of parameters may be changed when desired to learn more about the behavior of the system. Examples of such parameters existing in the model are the amount of real memory available to a task, the size of the buffer area, and the limits on the length of time a task may continuously use the CPU. These values are specified as system attributes in the SIMSCRIPT program.

Each program to be processes is a temporary entity with several attributes such as size of program, number of I/O requests, and processing time. The distribution of program size is obtained from real data and is given as a step function of 512-word blocks. Other parameters are similarly obtained and represented, and values of these attributes are generated randomly from their distributions.

Several events and subroutines are represented in the model for the operating system. A SIMSCRIPT timing routine permits the occurrence of both endogenous and exogenous events. These include

(1) the interruption of a task currently using the CPU,

(2) the completion of an I/O activity,

(3) the bringing of new tasks into core to compete for the use of the CPU and I/O facilities, and

(4) the completion of a task.

For the purpose of meansuring the system performance and summarizing its behavior, statistics are gathered at intervals of the simulated time. These include CPU utilization in the time interval, accumulated CPU productivity, CPU idle and busy time. Other information such as mean program sizes, mean compute time, mean I/O time, number of I/O requests, and number of jobs finished are also gathered.

The validity of the model is improved in two ways. First, in the model-building phase, each distribution is tested for goodness-of-fit with the actual distribution. Second, simulation runs using real workload data are compared with data on performance of the real system.

Experimental Results

The algorithm for finding the optimal buffer size given earlier is programmed in FORTRAN. From the five sample days' runs, the optimal sizes range from 12 to 18 sectors at a mean of about 15. This is slightly less than twice the 224-word buffer used on the UNIVAC 1108 under study. The reason for the variation of the optimal buffer size for each day's run is that this is determined based on parameters such as average program length and number of I/O requests which change dynamically for each day. A sensitivity analysis performed through the simulation model would better show how the optimal buffer size changes as various parameters are changed.

In the simulation model, runs are initially made using only buffer sizes 8 and 16 sectors to investigate (1) the length of the initialization interval necessary for stabilization of the system, and (2) the length of the run segments at the end of which, statistics are gathered.

Since parameters of the workload are described by distributions and their values determined by random look-up procedures, simulation runs are repeated for the same basic model using a different seed for generating random numbers. Simulation results are obtained for buffer sizes ranging from 8 to 24 sectors with varied workload characteristics.

In order to see how the system performs with different buffer sizes as various parameters (or their distributions) are changed, simulation runs are also made where only one parameter at a time is varied from the basic model. This is done with the distributions for CPU time and the number of I/O requests of individual programs. Of the buffer sizes used, that of 12 sectors or 336 words generally gives the best system performance.

The difference between the optimum size obtained by the algorithm and that of the simulation study is expected. The distribution of buffer size versus CPU productivity obtained through the algorithm has shown each local peaks to be highly sensitive to small increases of the buffer size. For this reason, a slightly smaller buffer size than the mathematically optimal solution should yield better performance.

CONCLUSION

In the case analysis, the results of the many simulation runs are obtained at high cost. If solutions to the complementary analytic model were considered, the numerous alternatives to be tested would have been substantially decreased. Clearly in this particular application, solutions to the mathematical programming model would have served this purpose.

In considering more complex system design and selection problems such as memory hierarchy optimization, the experimental design problem of selecting the factors and the levels of each factor to be tested becomes even more acute. Solutions to mathematical programming models, appropriately formulated, would help in reducing the alternatives at substantially low cost. Furthermore, the approach used in the solution procedures discussed here provides additional useful information.

REFERENCES

1. Lucas, H. C., Jr., "Performance Evaluation and Monitoring," Computing Surveys, 3, September, 1971.

2. Naylor, T. H., Burdick, D. S., and Sasser, W. E., Jr., The Design of Computer Simulation Experiments, Duke University Press, Durham, 1969, 1-35.

3. Kho, J. W., and Pinkerton, T. B., Optimal I/O Buffer Size for Multiprogramming, Computer Science Technical Report 107, University of Wisconsin, Jan., 1971.

4. Kho, J. W., Optimal Organization of I/O Operations in Multiprogrammed Systems, Ph.D. Dissertation, University of Wisconsin, Dec., 1972.

5. Martin, J., Design of Real-Time Computer Systems, Prentice-Hall, Englewood Cliffs, 1967, 315-316.

6. Hu, T. C., Integer Programming and Network Flows, Addison-Wesley, Reading, 1969.