The performance of different features of system software can be compared efficiently by means of rapid, on-line switching between the versions. This technique of on-line switching has been applied to determine the effects of page replacement algorithm, time-slice length, and user priority setting in the CP-67 *time-sharing system.*

Experimental evaluation of system performance

by Y. Bard

Developers of computer systems are frequently faced with a choice among several possible implementations of a given function. Similary, the operator of a computer system faces a choice in setting adjustable parameters built into the system. In each case, the alternative that optimizes system performance is sought. While analytic or simulation models may sometimes assist in making the choice, it is true that in most cases only actual running tests of the various alternatives can provide definitive answers. But even when all the variations can be tested on a real system, it is not easy to obtain meaningful results. One has the choice between performing the tests on a carefully controlled benchmark job stream, or on the system in an actual uncontrolled working environment. In the first case, the results may be inapplicable except to the specific benchmarks chosen, and a realistic benchmark stream is often hard to come by (this is particularly true for time-sharing or real-time computer systems). In the second case, random fluctuations in the magnitude and nature of the load placed on the system make it difficult to compare the results obtained under different system versions. Consequently, although performance evaluation is still possible, data must be gathered over a fairly long period of time. For instance, different versions of the CP-67 time-sharing system[1,2] were compared by Bard[3] under actual operating conditions. However, several months' worth of data were required. To meet the objectives outlined above, results must be forthcoming at a faster pace.

If two versions of a system are to be compared, it is desirable to run them alternately in some pattern that would compensate for the random load fluctuations. One such experiment has been described by Margolin, Parmelee, and Schatzoff.[4] The purpose of that experiment was to compare two versions of the CP-67 system that differed in the method of handling free storage re-

quirements. A system incorporating the old free storage algorithm was run on Monday and Thursday of one week and on Tuesday and Wednesday of the next week. A new free storage algorithm was run on Tuesday and Wednesday of the first week and Monday and Thursday of the second week. Measurements taken in the course of these two weeks revealed a considerable reduction in CPU overhead required by the new algorithm. The new version also seemed to account for a slight increase in useful throughput (e.g., problem state CPU utilization, and user I/O rates), but the significance of the increase was open to doubt. It is our purpose here to show how on-line experimentation can produce significant results in shorter periods of time.

Performance optimization

The stated objective of conducting experiments is to optimize the system's performance. This cannot, in principle, be done unless one defines a performance objective function whose value is to be minimized or maximized. Unfortunately, one is usually interested in several different aspects of performance, and a single objective function is hard to come by. It is not our intention to prescribe such an objective function here. Rather, we show how the results of the same set of experiments can be evaluated in the light of several different performance criteria, such as throughput rates and response time. Thus, even though different performance criteria may be thought appropriate in different cases, the experimental technique remains valid.

Even if the performance criterion is well-defined, optimization is rarely achieved in one experiment that compares two versions or several settings of a small number of parameters. Such an experiment is usually only one stage in a series of experiments, in which a larger number of parameters are varied over wider ranges. This paper concerns itself primarily with a technique for conducting each *single* experiment in the series. Techniques for designing a series of experiments are briefly alluded to in the section on extensions at the end of this paper.

On-line experimentation

To achieve the desired objectives, it is usually required to compare the performance of two or more versions of the system. These versions may differ simply in the settings of some parameters (e.g., time-slice length), or may contain different algorithms for performing certain functions (e.g., page selection or task dispatching). The time required to obtain significant measures of performance differences between these versions is minimized if the loads under which these versions are tested are

Table 1 DUSETIMR wait time between observations*

Number of logged-on users	*DUSETIMR wait time (sec)*
10 or less	1000
11	851
12	724
13	617
14	525
15	447
16	380
17	324
18	275
19	234
20	100
21	170
22	144
23	123
24	105
25	89
26	76
27	64
28 or more	60

*Does not include DUSETIMR running time

equalized. This can be achieved if one switches rapidly between the versions under consideration. If the versions could be alternated every few minutes, say, then there would be little likelihood that version I sees, on the average, a consistently heavier or lighter load than version II. Since it is impractical to shut down one version and load the other one every few minutes, it is necessary to code the system in such a way that the versions can be switched on the fly, without impairing any system functions. This is usually possible when versions differ in the settings of parameters: it is only necessary to store the desired values in the appropriate locations. When different algorithms are to be compared, it is necessary to assemble all of them into the system and to provide a switch into which a transfer to the desired version can be stored. To automate the experiment, the following setup is suggested: It is assumed that a software monitor that "wakes" up periodically and collects system performance data is available.

The program that collects the measurements should be given the privilege of storing parameter or switch values into the system. In this way, the switching of versions can be synchronized with the collection of data:

1. Store the initial parameter value.
2. Take several observations at the usual intervals.
3. Store the alternate parameter value.
4. Repeat step 2.
5. Return to step 1.

In deciding how many observations should be taken between switches, the following considerations arise: The switching from one version to another may give rise to transient effects, and data taken before these effects have died out should be discarded. For instance, suppose observations are to be made at one-minute intervals. The two versions under consideration differ in the page replacement algorithm (see Example 1 below). We know that the system reads pages at a rate of about 50 per second and that main storage holds about 100 user pages. Therefore, the average page residence time is two seconds, and only the first observation taken after a switch is likely to be affected. To achieve the best load balancing, it is desirable to switch as often as possible, which in this case would mean once every two observations. As a result, however, 50 percent of all observations would be discarded. Usually, a batch size somewhat in excess of two observations yields the best compromise between losses of information due to (1) discarded data, and (2) load fluctuation. A method for determining the optimal batch size and the required total number of observations is described elsewhere.[5]

Table 2 Synthetic benchmark characteristics

Benchmark designation	*Virtual CPU time (msec)*	*Virtual SIO operations to disk*	*Pages referenced**
Trivial	1	0	1
Mixed	260	75	20
I/O-bound	560	225	12
CPU-bound	690	0	1

*Exclusive of I/O routines

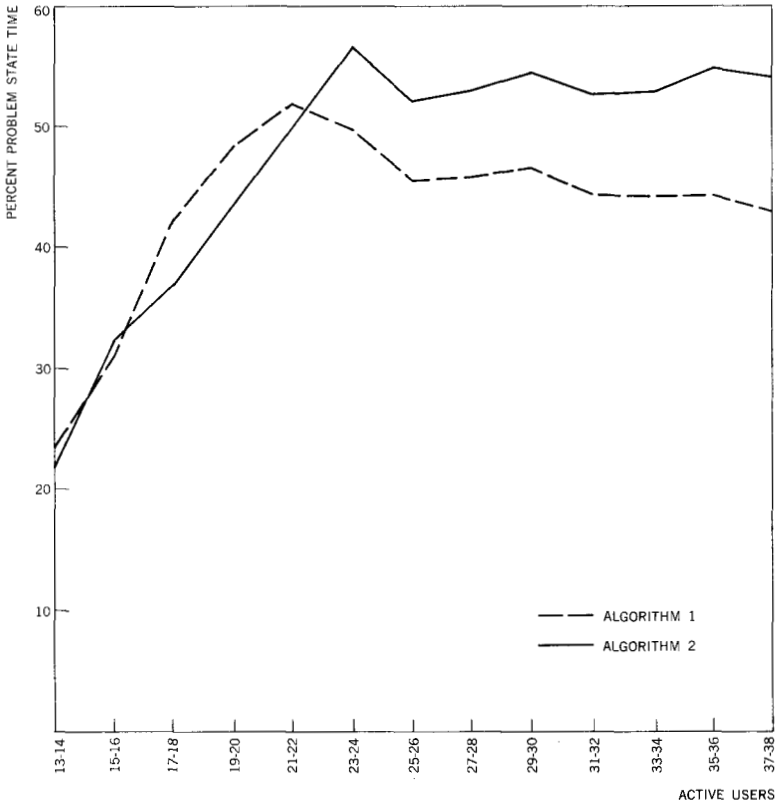
The proposed technique is obviously not applicable in all situations. The system variations that are to be compared may be so incompatible that one simply cannot switch between them on the fly. Or, the transient effects may be so long that switching can only occur very infrequently. The power of the technique is so strong, however, that it will often be worthwhile to plan and implement the various versions specifically so as to permit rapid switching.

Data collection and reduction

We describe below three experiments that were run on a CP-67 system[1,6] at the Cambridge Scientific Center. The performance of the system is monitored around the clock by a program called DUSETIMR (see Bard[3]). This program remains dormant for a specified time period that depends on the number of users on the system (see Table 1). It then "wakes up" and runs a set of four synthetic benchmarks[7] which are designed to exercise system resources in given proportions (see Table 2). The running time of each benchmark is recorded, as are the contents of various counters maintained by CP-67. The system was instrumented in such a way that on every nth running, DUSETIMR could store a value that effected the switch from one version to another. The three examples given here illustrate the use of the method for choosing between system algorithm implementations, for tuning adjustable parameters, and for evaluating the effects of priority assignments.

The data reduction procedures used to obtain the results described in the examples are quite simple. Suppose two versions of the system are being compared. The observations are sorted into two subsets, depending on the version in effect at the time of the observation. If necessary, the first observation(s) from each batch is discarded. Each one of the two data sets is then analyzed separately. For instance, to produce Figure 1, each data set is further divided into subsets, the kth subset containing those observations in which the number of active users[8] is $2k$-1 or $2k$. The average value of percent problem state time is now computed for each subset. The averages from both sets are then

Figure 1 Comparison between page replacement algorithms—CPU



plotted against the number of active users, producing the two curves of Figure 1. To produce Figure 3, the original sets were again broken up into subsets corresponding to the number of active users, but the subdivision was coarser. The distribution of mixed benchmark completion times was determined within each subset, and the 75 percentile of that distribution was plotted against the number of active users.

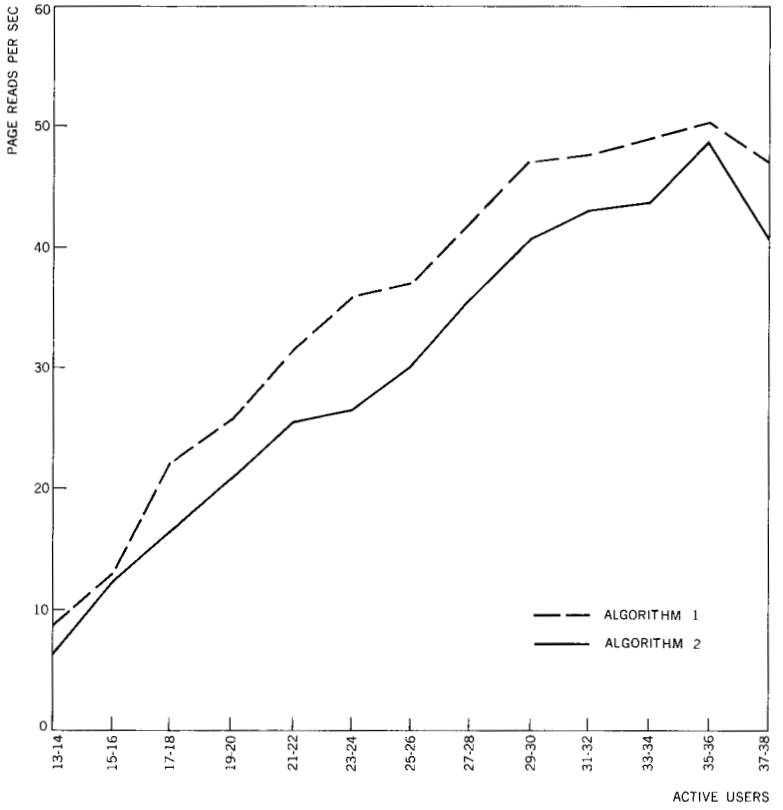
To obtain Figure 4, those observations were extracted (from each set) that were not exceeded by any other observation in the values of both virtual disk I/O operations per second and percent problem state time. These so-called "Pareto-maximal" observations are plotted for both sets. The curves joining these points represent the limits of the observed operating regions of the two system versions.

Examples

page replacement algorithm

The purpose of the first experiment was to select the better of two page replacement algorithms. A page replacement algorithm is a method for choosing which user page should be removed

Figure 2 Comparison between page replacement algorithms—paging

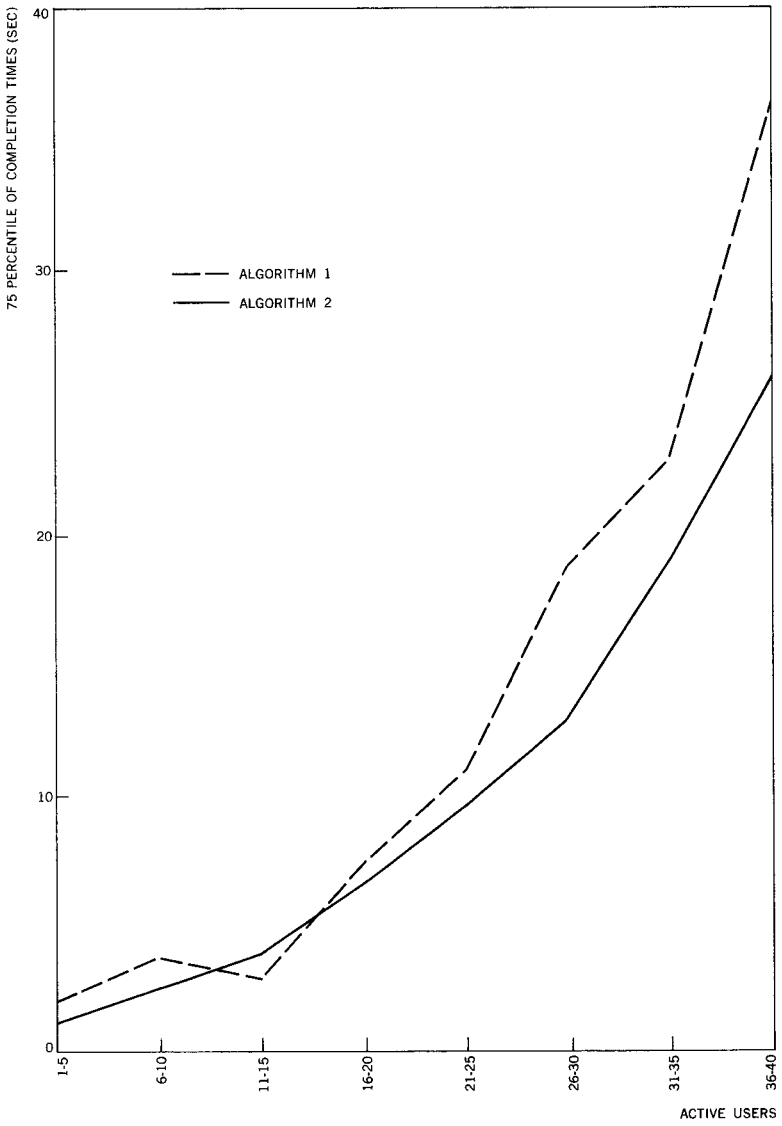


from main storage to make room for a new page that is to be brought in from auxiliary storage at a user's request. The two algorithms are described in Item 1 of the Appendix.

For this experiment, a batch size of five observations was chosen. As noted above, the average page residence time is about two seconds. Hence, transient effects were judged to be short, and only the first observation in each batch needed to be discarded. The experiment was run for five days, and some of the results are plotted in Figures 1 through 4. It is worth noting that the results for each day when taken separately were almost as conclusive.

Figure 1 shows that as soon as the number of active users exceeds 22, Algorithm 2 delivers up to 10 percent more CPU time to the users. Figure 2 shows that this increase in CPU utilization is actually obtained with a decrease in paging activity. Figure 3 shows that Algorithm 2 considerably improves the completion time of the mixed benchmark. Similar improvements were noted for the trivial and I/O-bound benchmarks. One verifies easily that the probability of obtaining such results if Algorithm 2 is not, in fact, superior to Algorithm 1 is practically nil. Hence, at

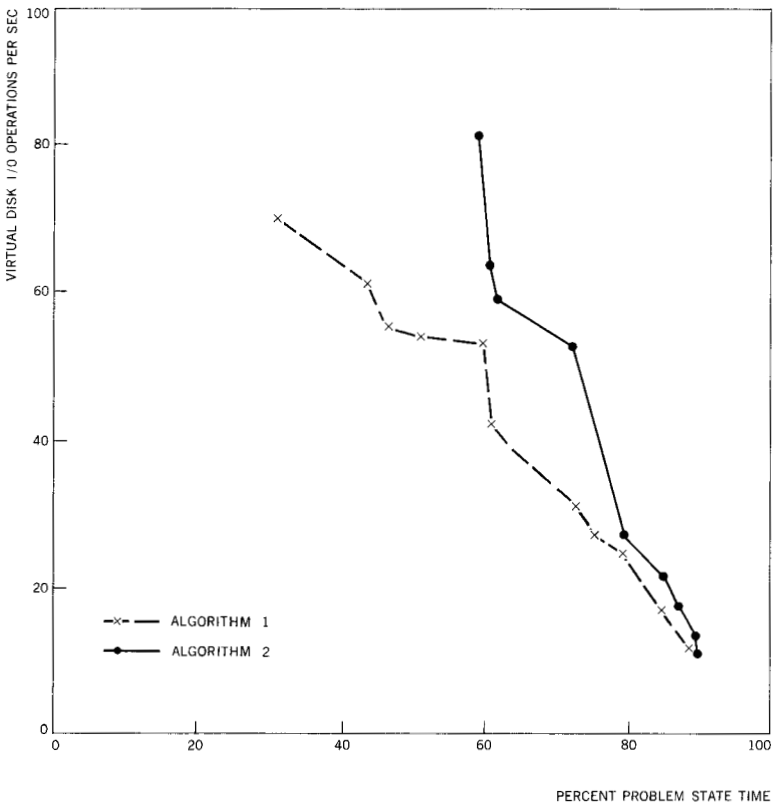
Figure 3 Comparison between page replacement algorithms—mixed benchmark completion times



least under the load conditions encountered in the course of this experiment, the superiority of Algorithm 2 is established beyond doubt.

An additional illustration of the superiority of Algorithm 2 is given in Figure 4, which shows the Pareto-maximal observations on the variables problem state and user disk I/O operations per second. These points represent the outer limits of the system's observed operating region. The larger extent of the operating

Figure 4 Comparison between page replacement algorithms—boundary of operating region



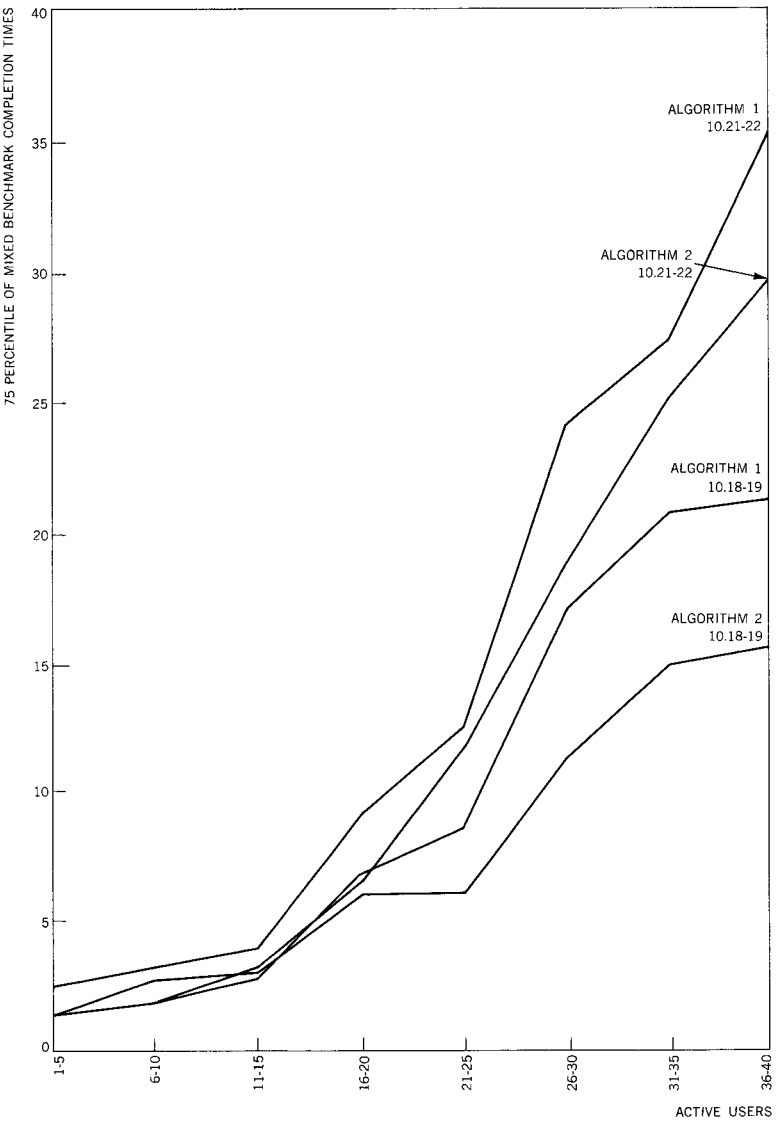
region under Algorithm 2 suggests that the system's capacity for performing useful work has been increased.

Figure 5 clearly illustrates the power of the switching technique. It might have been argued that the superiority of Algorithm 2 was so clear that any measurement technique would have sufficed. It so happened, however, that on two of the days of the experiment (October 18–19, 1971) system response was relatively fast, whereas on two other days (October 21–22) the response (with same number of active users) was relatively slow. Had the experiment been conducted by running Algorithm 1 on the first two days and Algorithm 2 on the last two days, then, as shown in Figure 5, one would have concluded that Algorithm 1 provided the faster response time!

time slice

The aim of the next experiment was to test the effect of the interactive and noninteractive (see Appendix, Item 2) time-slice lengths on system performance. Two values were chosen for each time slice: $t_1 = 200$ and 300 milliseconds, and $t_2 = 3000$ and

Figure 5 Page replacement algorithms—comparison of results by day



5000 milliseconds. Each cycle in the experiment consisted of 20 observations, including batches of five observations taken under each one of the four possible combinations of time-slice values. A pseudo-random number generator was used to vary the order in which the four combinations occurred within each cycle. Transient effects were judged to be short, and consequently only the first observation was discarded from each batch. The experiment was run over a period of one week.

Some of the results are shown in Figure 6, which displays the 75 percentile of the mixed-benchmark completion times. Here, the

values $t_1 = 300$ and $t_2 = 3000$ seem to be superior. Otherwise, it appeared that system performance on the whole was quite insensitive to time-slice lengths within the range of tested values.

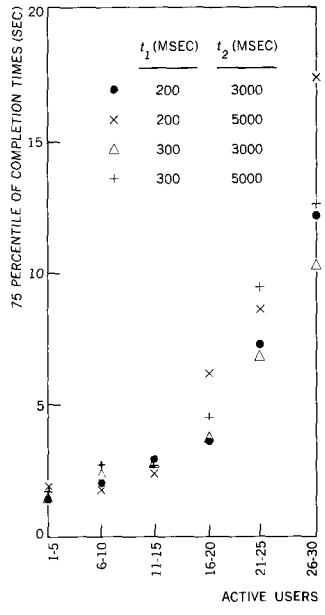
The values used in CP-67 version 3.1 are actually the ones that favor our mixed benchmark, which is similar in structure to a very short compilation or assembly.

user priority

The aim of the third experiment was to determine how a specific user's performance would be affected by his assigned priority P_0 (see Appendix, Item 3). For the purposes of the experiment, the DUSETIMR program was allowed to set its own value of P_0 prior to running its set of benchmarks.

In the course of the experiment, successive observations were taken with P_0 cycling through the values 1, 50, and 98. The results are shown in Figure 7. It appears that P_0 has only a slight effect on response when there are no more than 25 active users on the system, but the effect becomes quite spectacular beyond that point. An essentially even response is assured to the user with priority level 1, while the average user ($P_0 = 50$) suffers considerable degradation and the $P_0 = 98$ user practically grinds to a standstill. Note that the 25 active-user level is also the one at which system throughput levels off (see Figure 1). Beyond this level, one user can buy improved performance only at the expense of other users.

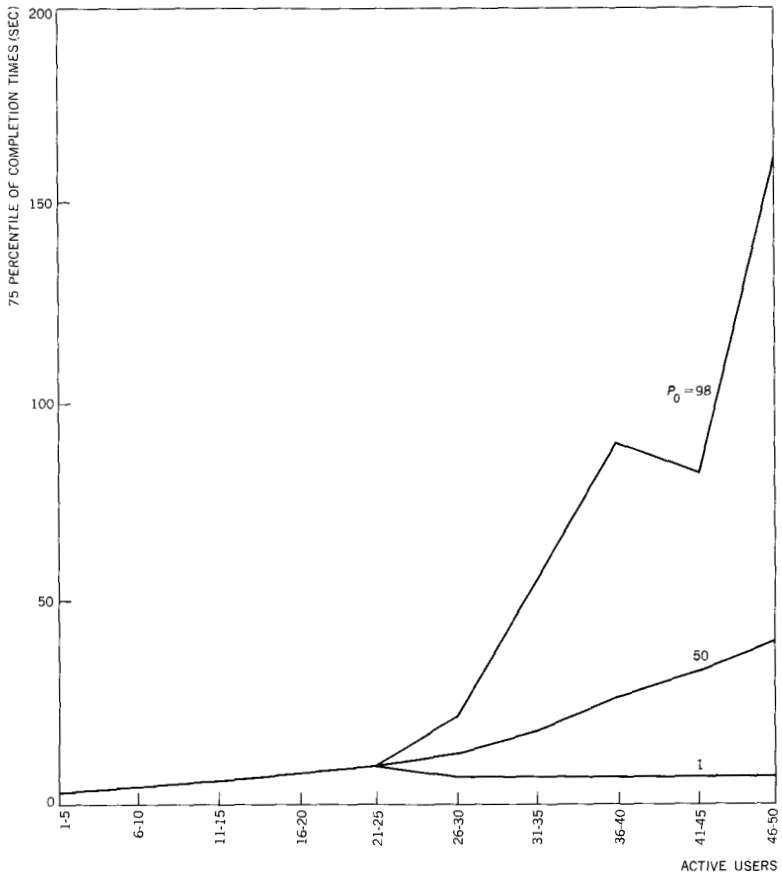
Figure 6 Time slice experiment — mixed benchmark completion times



Extensions

The experimental method described here can be useful as a system development tool by assisting the developer in choosing algorithms and parameter values. However, when one considers the diversity of applications that a general-purpose computer must serve, it becomes clear that no one algorithm or parameter setting will be optimal for all installations. It is then necessary to select the best settings—i.e., tune the system—for each installation. Furthermore, since the work load at any given installation may change with time, both in quality and quantity, it is necessary to retune the system periodically or continuously in an adaptive manner. The problem is similar to that of "tuning" a petroleum refining unit to the crude oil feed, or a chemical process to the raw chemicals that constitute its inputs. The EVOP (EVolutionary OPeration) technique,[9] which has been used extensively in the optimization of chemical and other industrial processes, may also be applicable to computer systems. In this technique, one selects a set of adjustable parameters, and one performs a sequence of experiments in which the values of these parameters are adjusted in a systematic manner within a well-defined operating region. The results are evaluated, the operat-

Figure 7 Effect of priority on mixed benchmark completion times



ing region is shifted in the direction indicated as most likely to improve performance, and a new series of experiments is run. The technique described in this paper would be used to conduct each individual experiment in the series.

In a stationary system, this cycle is repeated until the best operating region is located. In a system whose characteristics change with time, EVOP is carried on indefinitely in an effort to track the changing optimum conditions. While a fair amount of effort must go into the implementation of EVOP, it does provide a systematic manner for tuning the system, and this is particularly important when the system is so complex that intuitive prediction of the effects of various parameter changes is well nigh impossible.

ACKNOWLEDGMENTS

The author is indebted to Dr. M. Schatzoff for suggesting the idea of performing system evaluation experiments, to L. Wheeler for providing the switching mechanism in CP-67, and to M.

Fleming for placing the facilities of the Cambridge Scientific Center computer installation at our disposal for these experiments.

CITED REFERENCES AND FOOTNOTE

1. *CP-67/CMS System Description Manual*, Form No. GH20-0802-2, International Business Machines Corporation, Data Processing Division, White Plains, New York (1971).
2. R. A. Meyer and L. H. Seawright, "A virtual machine time-sharing system," *IBM Systems Journal* **9**, No. 3, 199–218 (1970).
3. Y. Bard, "Performance criteria and measurement for a time-sharing system," *IBM Systems Journal* **10**, No. 3, 193–216 (1971).
4. B. H. Margolin, R. P. Parmelee, and M. Schatzoff, "Analysis of free storage algorithms," *IBM Systems Journal* **10**, No. 4, 283–304 (1971).
5. Y. Bard, *A Technique for Performance Comparison of System Software Features*, IBM Cambridge Scientific Center Technical Report No. 320–2081, Cambridge, Massachusetts (1972).
6. *CP-67 Program Logic Manual*, Form No. GY20-0590-1, International Business Machines Corporation, Data Processing Division, White Plains, New York (1971).
7. W. Buchholz, "A synthetic job for measuring system performance," *IBM Systems Journal* **8**, No. 4, 290–298 (1969).
8. An active user is one who has consumed some CPU time in the preceding observation period. Typically, at our installation, the number of active users was observed to be about two thirds of the number of signed-on users.
9. G. E. P. Box and N. R. Draper, *Evolutionary Operation*, John Wiley and Sons, New York, New York (1969).

Appendix

CP-67 maintains a table of all pages in main storage and a pointer that cycles around this table. CP-67 also maintains, at any given moment, a list of "in-queue" users, and these are the only ones eligible to receive service at that time. The manner in which this list is maintained is further discussed below. **Item 1**

The two algorithms may now be described as follows:

Algorithm 1: Move the pointer around the table until a page belonging to a user not in queue is found. If no such page is found in a complete trip around the table, select the next page for removal.

Algorithm 2: Select the next page if its reference bit is off. Otherwise, turn the reference bit off, move the pointer down one page, and repeat. Note: The reference bit is turned on whenever the user references the page in the course of running his program. The bit is turned off when the user is removed from in-queue status.

These algorithms correspond to those implemented in Versions 3.0 and 3.1 of CP-67, respectively.

Item 2 CP-67 separates the "in-queue" users into two classes: interactive users, who are said to be in Q_1, and noninteractive users, who are in Q_2. Roughly speaking, a user starting a new task is placed in Q_1, where he is allotted up to t_1 milliseconds of CPU time. If he is not finished, he becomes a Q_2 candidate. When CP-67 decides that there is room, the user is placed in Q_2 where he may receive up to t_2 milliseconds of CPU time. If not yet finished, he is made a Q_2 candidate again, and will be eligible for t_2 more milliseconds when next readmitted into Q_2, and so on until he is finished (or until there is an interruption from his console, whereupon he returns to Q_1). The quantities t_1 and t_2 are called the Q_1 and Q_2 *time slices*, respectively.

Item 3 A user's priority to enter Q_2 is determined by the following formula:

$$P = P_0 + 4T$$

where P_0 is a number between zero and 98 permanently assigned to the user by the installation manager, and T is the time (in seconds since system start-up) at which the user became a Q_2 candidate. When room in main storage allows, the candidate with lowest value of P is admitted. It is the practice at the Cambridge Scientific Center's CP-67 installation that practically all users are assigned the value $P_0 = 50$.