Operating Systems
B. Randell Editor

Trace-Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System

Stephen Sherman, Forest Baskett III, and J.C. Browne
The University of Texas at Austin

Microscopic level job stream data obtained in a production environment by an event-driven software probe is used to drive a model of a multiprogramming computer system. The CPU scheduling algorithm of the model is systematically varied. This technique, called trace-driven modeling, provides an accurate replica of a production environment for the testing of variations in the system. At the same time alterations in scheduling methods can be easily carried out in a controlled way with cause and effects relationships being isolated. The scheduling methods tested included the best possible and worst possible methods, the traditional methods of multiprogramming theory, round-robin, first-come-first-served, etc., and dynamic predictors. The relative and absolute performances of these scheduling methods are given. It is concluded that a successful CPU scheduling method must be preemptive and must prevent a given job from holding the CPU for too long a period.

Key Words and Phrases: scheduling, CPU scheduling, multiprogramming, performance measurement, trace driven models

CR Categories: 4.10, 4.20, 4.31, 4.32

This research was supported in part by the National Science Foundation under Grant GJ-1084. A preliminary version of this paper was given at the SIGOPS workshop on System Performance Evaluation, Cambridge, Mass. April 1971.

Introduction

The primary purpose of a multiprogrammed operating system is to bring into balance the demand for processing and I/O facilities presented by the user job load, so that computing facilities and I/O facilities are utilized to the ultimate degree in order to maximize throughput. Efforts to bring about a balance between demand for compute resources and I/O resources can be approached from a job scheduling (external) viewpoint, where efforts are made to load into the working memory a set of jobs which include both compute bound and I/O bound jobs. It is, however, well known that the character of resource demand of a given job may fluctuate dramatically over short-time intervals of operation. It is therefore necessary to consider internal processor, or CPU scheduling, and I/O facility scheduling if utilization of both classes of resources is to be efficient. Several papers [1, 2, 3, 4] have suggested that dynamic calculations of internal priorities for scheduling the CPU for multiprogrammed jobs will yield a significant improvement in throughput and CPU efficiency. All of these scheduling methods attempt to give higher priorities to I/O bound jobs than to CPU bound jobs, and to assume that the past performance of a job, either its immediate past or an average of its history, is a valid indicator of future performance. The basic concept of all of these papers is that priority of a given process for CPU service should be determined

Authors' addresses: S. Sherman and J.C. Browne, Department of Computer Science and Computation Center, The University of Texas at Austin, Austin, TX 78712; F. Baskett III, Department of Computer Science, Stanford University, Stanford, CA.

by the probability that the process will generate an I/O request in the next period of CPU activity. Each previously studied case, however, implicitly or explicitly involved alterations in the scheduling policy or other significant changes in the system in addition to dynamic assignment of CPU priority. None of the previous workers made any systematic effort to compare the various possible CPU scheduling methods. All of the previously reported work was done by altering the scheduling policy in an actual operating system. In these circumstances in order to attempt valid comparison of any great number of algorithms, a great deal of labor would be necessary.

This paper compares various CPU scheduling policies under circumstances such as: (1) the effects of the scheduling algorithms are isolated and valid comparisons [5] between different policies can be made, and (2) changes in the CPU scheduling methods become very easy to make. We use a trace-driven model, TDM, as defined by Cheng [6] as the test-bed for comparison of the scheduling methods. (See also Pinkerton [7].) The data to drive the TDM was obtained in a separate study [8] where a software probe recorded the activity of the UT Austin CDC 6600 system at the minimum quantum level of system activity. This data, including such factors as CPU burst times and I/O service times, was used to produce a job stream whose demands for CPU and I/O service are known at the microscopic level. This job stream is then run through the model system with a set of CPU scheduling algorithms. In addition to considering several possible dynamic predictors for CPU scheduling we use the standard algorithms of multiprogramming theory, round robin, first-come-first-served, etc., as well as theoretic "best" and "worst" scheduling policies. Several surprising results are obtained. In particular, it becomes clear that essential factors in a successful scheduling algorithm are a preemptive mechanism and a bound on the CPU burst time, either directly by imposing an upper bound or indirectly via a round-robin method.

Previous Work

Most of the previous experimental work published in this problem area has been concerned with the use of dynamic predictors. Stevens [1] used the running average of compute time to I/O time charged to each job as the measure of internal priority for CPU scheduling. He observed a doubling of throughput and an increase of 18 percent in CPU utilization. However, a number of other changes were made in the system at the same time. It is thus difficult to totally isolate effectiveness of the dynamic scheduling in this case. Marshall [2] considered two schemes. One was a reward-penalize scheme in which, if a job requested I/O before the termination of a system quantum of time, it received a longer quantum of activity the next time it was run, whereas, if it failed to generate an I/O activity in a minimum system quantum, it was penalized and given a smaller quantum the next time it was run. This did not turn out to be a particularly significant approach. Marshall then went to a cumulative ratio as a predictor. He used the ratio of total wait time over wait time plus CPU time as an internal priority measure. This was found to be an effective procedure which increased productivity by a significant amount in actual operation. Marshall, however, imposed a maximum burst time of 5 sec by forcing a rescheduling of internal priorities at this interval. We shall see later that this should probably have a very significant effect on the performance metrics. Wulf [3] has studied dynamic internal scheduling and has implemented a moving average procedure on the Burrough's B5500. He found that this modification in scheduling together with changing the job mix produced a 25 percent increase in processor utilization. However, it is not clear what improvement is due to the dynamic priority adjustment. Ryder [4] implemented a scheme basically utilizing the information gathered in the last time quantum of job operation to predict the future behavior. Ryder's algorithm was essentially heuristic and had six factors, including a reward-penalize factor and an upper-bound on CPU burst time. It is difficult to isolate from his algorithm the key factors which yield improvement in CPU utilization and throughput. Ryder also made some effort to correlate his results with CPU-I/O overlap, a subject which we will discuss in a subsequent paper. Baskett, Raike, and Browne [9] and Schwetman and Browne [10] have studied the use of round-robin procedures with varying time quantum size.

The Trace-Driven Model

The data used in this study was collected on a CDC 6600 running under the UT-1 operating system by the use of an event-driven software probe [8]. The CDC 6600 has 128 k words of central core memory, 4-6638 disks, 12 I/O channels, and one half million words of extended core storage. The 6600 itself is an 11 computer complex consisting of a very fast central processor (CPU) and 10 peripheral processors, PPU. The PPUs are primarily used for I/O and control functions. The UT-1 system under consideration designates one PPU as the monitor to coordinate all system activity. One PPU is used to drive the operator display console. Three other PPUs are dedicated to other tasks such as, driving the teletype system, driving the card readers, line printer, etc., and driving the high speed remote computers. The remaining 5 PPUs are available to perform transit system functions and user I/O. The 128 k words of central memory are allocated (by software) to seven or fewer control points which are virtual CPU's. Only one program at a time can be assigned to a control point, and a program is not a candidate for execution

Table I. Comparison of Model Results and Actual Results

	RUN 1	RUN 1 MODELED	RUN 2	RUN 2 MODELED
Time of day	10:13:42	—	20:24:16	—
Jobs started	547	—	275	—
Jobs completed	552	—	270	—
Elapsed time*	4037.35	4048.47	2435.23	2434.26
CPU active time*	3639.49	3653.25	2318.06	2316.07
Percent active*	90.15	90.24	95.19	95.15
User PPI idle time*	2271.15	2198.52	1254.71	1518.83
Percent idle	56.27	54.30	51.56	62.39

DEGREE OF MULTIPROGRAMMING†

	Measured	Control var	Measured	Control var
Mean	5.436	5.436	5.915	5.915
Standard deviation	.997	.997	.899	.899

* All times are expressed in seconds.
† All seven control point streams were used in the validation.

in central memory unless it is assigned to a control point. The UT-1 system uses one control point for buffer space for all system I/O tasks. A second control point is used by the teletype network supervisor. This leaves five control points available for occupancy by user submitted jobs. All requests for the utilization of resources or for the initiation of new activity are handled by the monitor. The event-driven software probe is in the monitor. The probe records every request in a small buffer in central memory and dumps this information on tape. A careful analysis [8] has shown that the software probe has virtually no effect on system activity. A more detailed description of the CDC 6600, the UT-1 system, and the software probe can be found in Schwetman [8]. The empirical data used in this analysis was gathered on May 13, 1970. These tapes were selected for detailed analysis from a number of available tapes with highly similar characteristics. The system running time includes the initiation and completion of over 500 jobs for the first set of data and nearly 300 jobs for the second set of data. The two data sets reflect quite different external job mixes but extremely similar internal characteristics. This pattern is observed throughout the data sets obtained by Schwetman [8]. Two different runs are used to test the validity of our trace-driven model. The original data included all of the events requested and completed for seven control points and nine PPUs (the dedicated monitor is not included) as one stream of data. It was convenient to consider only the most pertinent requests due to the enormous volume of the original data. The following information was extracted from the original data tape for each control point with millisecond accuracy.

1. For each I/O request (requests for system service are included in I/O requests):
   1.1 The name of the request,
   1.2 the CPU time since the job started (the CPU time is needed since the UT-1 scheduler is RR),
   1.3 whether to give up the CPU or not.
2. The CPU time the job ended.
3. For each PPU function:
   3.1 the name of the PPU program (this corresponds to the name of the I/O request),
   3.2 the time the PPU requested that the CPU be taken away from the control point,
   3.3 the time the PPU finished.

The extracted information is in seven control point streams, one corresponding to each control point. The TDM will execute jobs taken from the seven streams and bring them into one executable stream again. The memory size required for each job was not recorded, but an analysis of the original data yields the mean and standard deviation of the degree of multiprogramming. This mean and standard deviation is used by the model to decide how many jobs should be running at one time. This control over average degree of multiprogramming was used in lieu of explicit memory management. The number of jobs in the trace data appeared to be sufficient to insure reproducible behavior when the order of scheduling was varied. Further, there is only a secondary control on the effects of possible channel and equipment request conflicts. This control consists of the increased length of the PPU process due to any conflicts. Another area ignored in the reduced data is the system time required by the central monitor to do certain functions such as storage compaction. However, the central monitor was active about 5 percent of the elapsed time, and the model takes this into account by allowing time for the central monitor on a rate basis. The amount of time that a job could overlap its own I/O was not available in the reduced data. The TDM assumed that a job could not use the CPU after an I/O operation had been requested. Fortunately, the difficulties presented above were either not very important or canceled each other out since the model was extremely accurate with respect to throughput and CPU utilization when simulating the two runs on which the data was collected using UT-1 scheduling methods. The utilization of PPUs was not as accurately returned on run 2 as on run 1. Since the PPUs were a surplus resource, the throughput and CPU utilization of the model was not sensitive to this variation. Table I illustrates the agreement of the results from the original runs and those given by the TDM. There is one other area which might have caused slight perturbations in the results. This concerns the problem of when to end the model. The model stops when it requires N control points to be occupied and there are less than N jobs available. The effect of this possible perturbation turned out to be very slight owing to the method for selecting the next job to occupy a control point. This method consisted of taking the next job from that control point stream for which the sum of the CPU time over all remaining jobs was largest. Once the accuracy of the model had been validated, the two special control points were dropped from consideration leaving only the five control points at which user submitted jobs were run with five PPUs made available to the users at the five control points. This was done because the special control points were not affected by the scheduling algorithm, and it is the effect of these algorithms which we want to consider in this paper. The system now represented by the model may be considered to be a system in which at most five jobs can be active at the same time and five I/O processes can occur simultaneously.

Results

Some of the statistical properties of the trace data were determined. The mean PPU disk service time was 86.32 ms with a standard deviation of 117.46 ms. The mean CPU service time between PPU requests was 77.15 ms with a standard deviation of 5638 ms. The median CPU service time was between 5 ms and 6 ms. The highly skewed characteristic of the CPU service time distribution appears to be a fairly general characteristic of multiprogrammed computer systems. It has been observed in many sets of measurements in this laboratory and in other studies [11, 12]. In order to test the significance of the contention that some jobs are "compute bound" and some jobs are "I/O bound," the first ten autocorrelation coefficients of CPU service times were calculated for each job. The mean of all CPU service times was used in the calculation. Although occasional jobs displayed significant autocorrelation coefficients, some were positive and some were negative. A weighted average was computed for each autocorrelation coefficient by weighting the corresponding coefficient for each job according to the number of CPU bursts per job. All the averages were positive but less than .01.

It has been widely stated and assumed that the "best" way to schedule the CPU in a multiprogramming system is to give the CPU to the job that will compute for the shortest period of time before issuing an I/O request. We accepted this and the dual assumption: the "worst" way to schedule the CPU is to give it to the job that will compute for the longest period of time before issuing and I/O request. Trace-driven modeling enables one to compute performance results for these two scheduling methods.

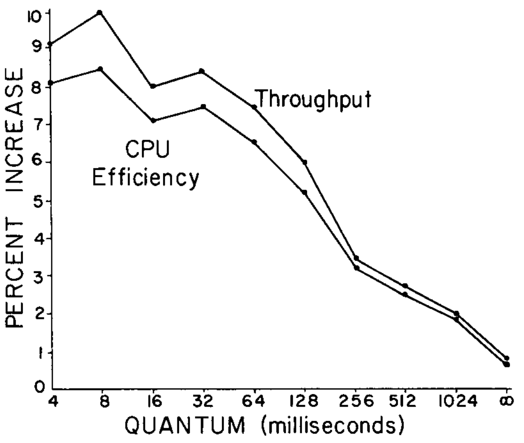
If the assumptions are correct then the performance results for all realistic scheduling methods will lie between these two extremes. Hence one can make absolute judgments about the performance of a scheduling method as well as relative judgments.

Two performance measures were computed for all CPU scheduling methods tested. The first measure is the total real time necessary to complete the trace data. We call this the throughput time. The second measure is the CPU efficiency as a percentage, that is, the total CPU time divided by the throughput time times 100. The results are presented as percentage increases over the "worst" method as follows: for scheduling method A, let T_A be the throughput time; let T_W be the throughput time for the worst method. The throughput increase for method A is $100^* (T_W - T_A)/T_W$. T_W was 4381.00 sec. If C_A is the CPU efficiency for method A, the CPU efficiency increase for method A is $C_A - C_W$ where C_W is 80.33%, the CPU efficiency for the worst method. The throughput increase for the "best" method over the worst method was 12.89%. The CPU efficiency increase was 11.19%. All other scheduling methods tested showed a positive throughput increase less than 12.89% and a positive CPU efficiency increase less than 11.19%. Thus no counter examples were found for the correctness of the best and worst assumptions. The high level of performance of the worst method in this resource-rich system accounts for the seemingly small increase. However, it should be noted that the best method recovers over half of the unused CPU time that occurs for the worst method.

The best and worst methods mentioned above are preemptive; that is, scheduling decisions are made whenever a job joins the CPU queue. The currently computing job may be stopped and later resumed at the point of interruption. The first experiment performed was to test *nonpreemptive* best and worst scheduling methods. In a nonpreemptive method, scheduling decisions are made only when the currently computing job issues in I/O request. The throughput increase for the "nonpreemptive best" method was 3.06% and the CPU efficiency increase 2.43%. The throughput increase for the "nonpreemptive worst" method was .20% and the CPU efficiency increase was .23%. Since the nonpreemptive best performance was so far below the preemptive best performance, it was decided that preemption is a necessary part of any good scheduling method and only one other nonpreemptive method was considered. It was the classical "First-Come-First-Served," FCFS, method. The throughput increase for FCFS was .78%, and the CPU efficiency increase was .68%. Note that this result lies between worst and best nonpreemptive methods.

Some queuing theory results (Baskett [13]) indicate that round-robin, RR, CPU scheduling will yield performance improvements when the CPU service time distribution is highly skewed and the autocorrelation coefficients are negligible. The next experiment was an experimental verification of the theoretical results.

Fig. 1. Throughput and CPU efficiency increase for round-robin scheduling.



RR CPU scheduling allows each job in the CPU queue to compute for a maximum of Q time units. If the currently computing job uses all Q time units without issuing an I/O request, that job is preempted and put at the end of the CPU queue. The job at the front of the CPU queue is then allowed to compute for a maximum of Q time units, etc. Q is called the quantum. Figure 1 shows the throughput increase and CPU efficiency increase for quantum sizes from 4 msec to 1024 msec in powers of 2 and a quantum size of infinity (FCFS scheduling). The experimental results have the form predicted by the theoretical results. The 8-msec quantum size gave the best performance with a throughput increase of 10.08% and a CPU efficiency increase of 8.75%. It should be noted that the model assumes no overhead is required to switch the CPU from one job to another. (On a CDC 6600, the overhead is actually 32 μsec.) Figure 1 also illustrates the degree to which throughput increases and CPU efficiency increases correspond. A similar degree of correspondence was found in all other scheduling methods tested.

The last experiment was concerned with preemptive predictive scheduling in the manner of Stevens [1], Marshall [2], Wulf [3], and Ryder [4]. The basic technique is to predict the length of the next CPU service time for a job based on the job's past behavior and to give the CPU to the job with the smallest predicted value. If the predictions are 100 percent accurate, this scheduling technique will be the "best" method. Six different scheduling methods of this type were tested. The first four methods use an "exponential smoothing" predictor (Brown [14]) as follows. If X_{n-1} is the $(n - 1)$-st CPU service time for a job and $\hat{X}_{n-1}$ is the $(n - 1)$-st prediction for that job then the prediction of X_n is

$$\hat{X}_n = \alpha X_{n-1} + (1 - \alpha)\hat{X}_{n-1} \tag{1}$$

where α is a real number between 0 and 1. The larger the value of α the more heavily weighted is the most recent past and the less heavily weighted is the more distant past. For $\alpha = 1$, the predicted next value is exactly equal to the most recent actual value. In general

$$\hat{X}_n = \alpha \sum_{i=0}^{n-1} (1 - \alpha)^i \hat{X}_{n-i-1}.$$

In these tests, it is assumed that $\hat{X}_0 = 0$ and that X_0 is the first service time.

Four different values of α determined four different prediction formulas. The values tested were $\alpha = 1$, $\alpha = .75$, $\alpha = .5$, and $\alpha = .25$. Since these scheduling methods were preemptive, a new scheduling decision was made each time a job joined the CPU queue (completed an I/O service). A new predicted CPU service time was determined for the job joining the CPU queue according to formula (1). A new predicted CPU service time was also determined for the job currently computing, which was the old predicted value minus the time already used. Note that this may lead to a negative predicted CPU service time. Then the CPU was given to the job with the smallest predicted value. For these tests the best results were achieved for $\alpha = .5$ with a 7.43% increase in throughput and 6.11% increase in CPU efficiency. The worst results were achieved for $\alpha = .25$ with a 6.66% increase in throughput and a 5.45% increase in CPU efficiency. It was noted that the results were good but not exceptional and that the differences in performance for different values of α were small. The percentage of correct choices made by these predictive methods was determined, and all four were close to 75%. When there was only one job in the CPU queue, the scheduling of the job was counted as a correct choice. This accounted for nearly half of the correct choices.

Two other predictives schedulers were tested. One was a "complete history" method and the other was a "random guess" method. The complete history method predicts that the next CPU service time will be equal to the mean of all past service times for that job. The formula is

$$\hat{X}_n = (X_{n-1} + \hat{X}_{n-1}(n - 1))/n \tag{2}$$

where $\hat{X}_0 = 0$ is assumed and X_0 is the first service time. This method is similar to that used by Stevens. The results for the complete history predictive method were in the same range as those for the exponential smoothing predictive methods, although the percentage of correct choices was slightly lower (73%). The random guess method gave an increase of 3.59% in throughput and 2.88% in CPU efficiency. The percentage of correct choices was also substantially lower (67%).

Table II. Summary of Performance of CPU Scheduling Methods

	BEST		WORST		FCFS
	Preemptive	Nonpreemptive	Preemptive	Nonpreemptive	
Thru	12.89	3.06	0	.20	.78
CPU	11.19	2.43	0	.23	.68

	Round robin								
Quantum	4	8	16	32	64	128	256	512	1024
Thru	9.04	10.08	8.09	8.52	7.45	5.99	3.52	2.64	1.87
CPU	8.02	8.75	7.16	7.55	6.63	5.23	3.27	2.47	1.76

	Predictive											
	Bound = 256 ms			Bound = 512 ms			Bound = 1024 ms			Unbounded		
	Thru	CPU	HITS	Thru	CPU	Hits	Thru	CPU	Hits	Thru	CPU	Hits
$\alpha = 1.0$	10.88	9.26	77.0	11.53	9.88	77.9	11.25	9.60	78.2	6.87	5.62	74.5
$\alpha = .75$	10.11	8.54	77.4	11.43	9.74	77.7	10.62	9.01	78.6	7.01	5.74	75.9
$\alpha = .5$	10.93	9.32	77.4	10.93	9.28	76.4	11.26	9.61	78.1	7.43	6.11	75.8
$\alpha = .25$	11.50	9.84	77.1	11.30	9.61	77.1	9.96	8.39	77.1	6.66	5.45	75.2
Complete History	9.85	8.29	72.8	9.63	8.09	74.4	7.96	6.59	75.3	6.71	5.48	73.2
Random Guess	6.85	5.60	60.0	5.92	4.82	62.8	4.76	3.76	65.1	3.59	2.88	66.6

Thru: percentage increase in throughput over WORST method.
CPU: percentage increase in CPU efficiency over WORST method.
Hits: percentage of correct scheduling decisions.

Following the suggestion of Marshall [2] and Ryder [4], these six predictive schedulers were modified to place an upper bound on the amount of time any one job was allowed to compute. This effectively divides a very long CPU service time into a sequence of smaller service times, all but the last of which is equal in length to the upper bound. The prediction formulas (1) and (2) are such that they would never predict a service time longer than the upper bound under these circumstances. If the value of the upper bound is B, a new scheduling decision is made at least every B-time units. Three bounds were tested: $B = 1024$, 512, and 256 msec. All substantially improved the performance of the six predictive scheduling methods. For the exponential smoothing methods, when $B = 1024$ ms the results for $\alpha = .5$ and $\alpha = 1$ were the best of the six and were very close to each other. When $B = 512$ ms, the exponential smoothing method with $\alpha = 1$ was the best, and when $B = 256$ ms, the results for $\alpha = .25$ were best. All four of the exponential smoothing methods gave very similar results for each B, and no general improvement was obtained by dropping B from 512 to 256. The throughput results for the six predictive methods for the three bounded cases and the one unbounded case are illustrated in Figure 2. All the numerical results are summarized in Table II.

Conclusions

Figure 3 illustrates the comparative throughput performance of the best of each type of CPU scheduling method. It is clear that a good scheduling method must be preemptive and must prevent any job from capturing the CPU for too long a period of time. The performance of round-robin scheduling is probably better than most would expect, and the performance of predictive methods without bounds is probably worse than most would expect. In a system where "fairness" is important, such as time-sharing systems and some university systems, and the actual overhead for switching the CPU from one job to another is small, a round-robin scheduling method might be preferred. In a system where external priorities are important or the switching overhead is large, an exponential smoothing method might be preferred since external priorities could be used for tie breaking. The number of switches performed for the 8 ms round-robin case was 376,286, while the number of switches performed for the $\alpha = 1$, $B = 512$ ms exponential smoothing case was 73,201. It is clear that the bound on the predictive techniques is necessary because the accuracy of the predictors is not extraordinary and the bound gives them another chance when they make a poor choice. Any predictive

Fig. 2. Throughput increase for predictive scheduling.

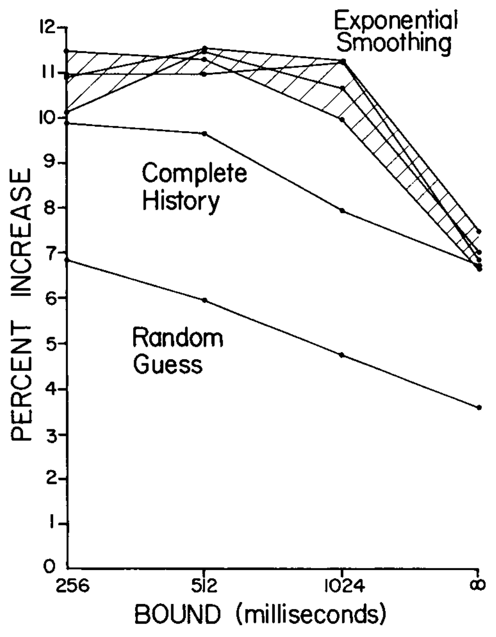
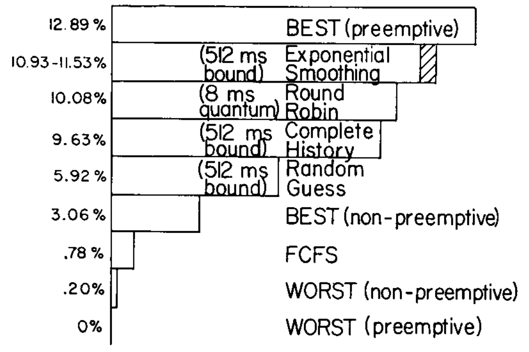


Fig. 3. Scheduling method versus throughput increase.



method can be expected to make a large number of poor choices since the autocorrelation coefficients for the CPU service times are so small.

Trace-driven modeling has been found to be an excellent vehicle for performing controlled scientific experiments to evaluate resource allocation policies in computer systems. "Live" experiments, in addition to being costly and time-consuming, suffer from an undetermined amount of noise in the results because of differences in the production environment and difficulty in isolating experimental variables. Traditional simulation models suffer from uncertain accuracy in the simulated input. The trace-driven model is flexible and practical. The results are accurate. These experiments have produced a better understanding of the factors contributing to successful CPU scheduling in a multiprogramming system.

Received February 1971; revised January 1972

References

1. Stevens, D.F. On overcoming high-priority paralysis in multiprogramming systems: A case history. *Comm. ACM 11*, 8 (Aug. 1968), 539–541.
2. Marshall, B.S. Dynamic calculation of dispatching priorities under OS/360 MVT. *Datamation* (Aug. 1969), 93–97.
3. Wulf, W.A. Performance monitors for multi-programming systems. Proc. Sec. Symp. Oper. Syst. Principles, Oct. 1969, 175–185.
4. Ryder, K.D. A heuristic approach to task dispatching. *IBM Syst. J. 8*, 3 (1970), 189–198.
5. Saltzer, J.H., and Gintell, J.W. The instrumentation of multics. *Comm. ACM 13*, 8 (Aug. 1970), 493–500. [These authors pungently draw attention to the need for controlled experimental studies.]
6. Cheng, P.S. Trace-driven system modeling. *IBM Syst. J. 8*, 4 (1969), 280–289.
7. Pinkerton, T.B. Performance monitoring in a time-sharing system. *Comm. ACM 12*, 8 (Nov. 1969), 608–610.
8. Schwetman, H.D. Jr. A study of resource utilization and performance evaluation of large-scale computer systems. Tech. Syst. Note-12, Computation Center, U. of Texas at Austin, 1970. (Also available as the Ph.D. dissertation of H.D. Schwetman, U. of Texas at Austin, 1970.)
9. Baskett, F., Browne, J.C., and Raike, W.M. The management of a multi-level non-paged memory system. Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J., pp. 459–465.
10. Schwetman, H.D., and Browne, J.C. Controlled experiments on the resource utilization on the multi-processor, multiprogrammed computer system. (Submitted for publication.)
11. Bryan, G.E. Joss: 20,000 hours at the console, a statistical summary. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, Montvale, N.J., pp. 769–777.
12. Scherr, A.L. *An Analysis of Time-Shared Computer Systems.* MIT Press, Cambridge, Mass., 1967.
13. Baskett, F. Mathematical models of computer systems, Ph.D. Diss., U. of Texas at Austin, 1970.
14. Brown, R.G. *Statistical Forecasting for Inventory Control.* McGraw-Hill, New York, 1959.