Y. Bard

Application of the Page Survival Index (PSI) to Virtual-memory System Performance

Abstract: The Page Survival Index (PSI) was defined in a preceding paper where it was used to describe the behavior of individual programs running in a time sharing environment. Here we show how a system-wide value of PSI can be calculated on the fly by the operating system. This value can be used to estimate users' memory requirements and to control system performance by maintaining the proper multiprogramming level. Simulation results show that a scheduler based on these concepts can achieve significant improvements in system performance.

1. Introduction

In this paper we are concerned with the performance of multiprogrammed virtual storage computer systems. We have shown previously [1, 2] that the effect of overall system activity on the behavior of an individual program running in such a system can be summarized by means of a parameter termed the page survival index (PSI). Furthermore, the PSI model made possible prediction of paging rates when a specific set of programs was run together. In the present paper, we show how overall system performance is related to the PSI and how direct control of the PSI can form the basis for efficient system scheduling.

The paper is organized as follows: In Section 2, we define the type of system to which our results apply. Sections 3 and 4 summarize the definitions and previous results relating to the PSI. In Section 5 we show how overall system PSI values can be calculated by the operating system, and in Section 6 we examine the performance of an actual system in relation to the PSI and other paging parameters. Section 7 shows that patterns established in individual program behavior can be generalized to the entire system.

Before going into the details of the suggested scheduler, we list some general principles for scheduler design (Section 8). In particular, the notions of feed-forward (Section 9) and feedback (Section 10) system control strategies are introduced. Simulation experiments demonstrating the potential usefulness and robustness of the proposed scheduler are described in Section 11. The Summary (Section 12) includes some remarks on scheduling, storage allocation, and system tuning.

2. System characterization

The systems to which this investigation applies may be characterized as follows:

- *Multiprogramming*

Several programs are allowed to share system resources. At intervals the system selects a subset of all programs requesting service, and only these are allowed to run. The number of selected programs is the *multiprogramming level* (MPL), and the selected programs are said to be *active*. The active set changes from time to time, as some active programs terminate or are deactivated by the system and new programs are activated. Service-requesting programs which are not currently active are said to be in the *eligible set*.

- *Virtual storage*

Each program has its own virtual address space which is mapped into real main storage in units of fixed size pages. A virtual page which has a current image in real main storage is called *resident*. Pages are brought into real main storage on demand, i.e., if a program references a nonresident virtual page, this page is fetched from secondary storage to replace some other currently resident page.

- *Page replacement algorithm*

The entire set of pageable main storage page frames is available to each active program. There is no discrimination among pages on the basis of ownership. The page to be replaced by an incoming page is determined by perusal of reference bits; pages not recently referenced are

replaced first. A typical implementation has a core table with an entry for each pageable page, and a pointer which cycles around this table. A page will be replaced when the pointer reaches it, if and only if it had not been referenced since the previous passage of the pointer. One complete round trip of the pointer is termed a *cycle*. It is easy to see [1] that a page will remain resident for at least one cycle and at most two cycles following the last reference to it. On the average, an unreferenced page will remain resident for 1.5 cycles before it is replaced.

The CP-67 [3] and VM/370 [4] systems are precisely of the type described here. Other virtual storage systems may be sufficiently similar to have our results applicable.

3. Program characterization

It has been suggested [1] that the behavior of a program running in a system as described in the previous section may best be described by a parameter called the PSI. The value of this parameter is defined as the number of interruptions in the course of program execution which an unreferenced resident page can survive before it is paged out. A high value of the PSI means that an unreferenced page can survive a long time, corresponding to periods of light system paging activity, and conversely.

From a program's page reference string the entire execution path (termed the Ψ-*realization*) of the program can be determined, on the assumption that the PSI maintains a constant value Ψ. From these paths (corresponding to different values of Ψ) one may compute various statistics, such as

- $A(\Psi)$ Average resident set size, i.e., average number of the program's pages resident in main storage.
- $R(\Psi)$ Average paging rate (page exceptions per second of CPU time).
- $t(\Psi)$ Average length of execution interval (seconds of CPU time between interruptions).
- $f(\Psi)$ Fraction of execution intervals terminating in page exceptions.

A plot of $R(\Psi)$ vs $A(\Psi)$, with Ψ as parameter, is termed the *paging characteristic* of the program. The paging characteristic may be considered an analog to the well-known parachor curve, which relates paging rate to storage allocation for the program running in a fixed storage partition. Plots given in [1] show the vast reduction in page exceptions which occurs when programs share storage dynamically, compared to fixed storage assignments.

4. Performance prediction for given active set

Suppose a given set of programs with known paging characteristics are to be activated together. It has been shown [1, 2] that under a round-robin dispatching policy, the same value of the PSI, say Ψ^*, applies to all the programs, and the average resident sets of the individual programs sum to the total available page frames:

$$\sum_i A_i(\Psi^*) = N, \tag{1}$$

where the subscript i identifies quantities characterizing the ith program. Furthermore, a complete round-robin cycle uses up, on the average, $\Sigma_i\, t_i(\Psi^*)$ seconds of CPU time, and during that period there will occur $\Sigma_i\, f_i(\Psi^*)$ page exceptions. Hence, the overall average paging rate (page exceptions per second of CPU time) is

$$P(\Psi^*) = \frac{\sum f_i(\Psi^*)}{\sum t_i(\Psi^*)}. \tag{2}$$

One verifies easily that $f_i = R_i t_i$, hence one may rewrite (2) as

$$P(\Psi^*) = \frac{1}{T(\Psi^*)} \sum_i t_i(\Psi^*)\ R_i\ (\Psi^*), \tag{3}$$

where

$$T(\Psi^*) = \sum_i t_i\ (\Psi^*).$$

To predict performance, one first solves equation (1) for Ψ^*, then uses (2) to compute P. Rates of other I/O types can be predicted similarly. It was shown previously [2] that this procedure works well (predicted paging rates are generally accurate within 10 percent) for a round-robin dispatcher and gives acceptable results (generally within 30 percent) for other dispatchers. Typical results are reproduced in Fig. 1.

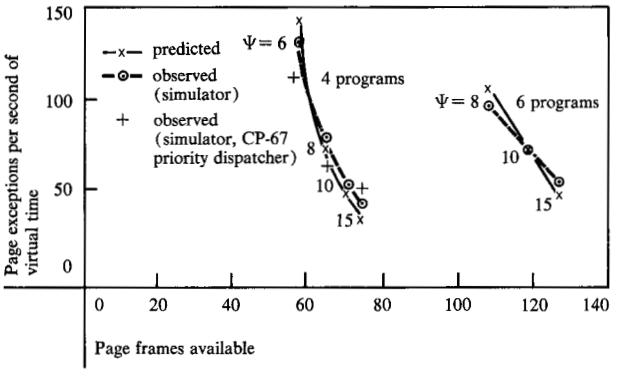


Figure 1 Prediction of system paging rate for a round-robin dispatcher.

5. Estimation of system PSI

The average system-wide PSI value over any sufficiently long (relative to page survival time) period may be de-

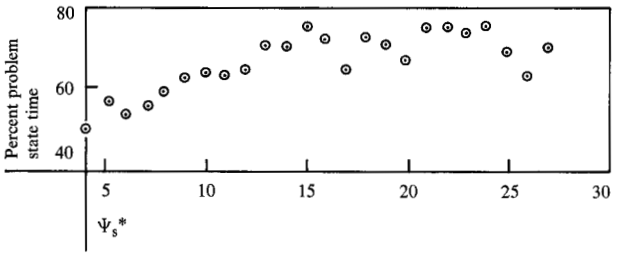


Figure 2 System performance vs page survival index.

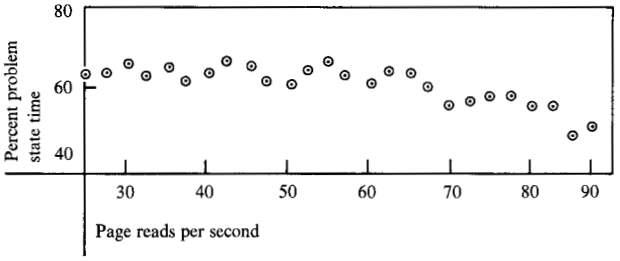


Figure 3 System performance vs paging rate.

fined as follows: Let $p_1, p_2, \cdots, p_m$ be the list of pages replaced by the system during that period. Suppose page p_j, belonging to program u_j, was replaced at time T_j, and suppose the last preceding reference to that page was made at time $t_j < T_j$. Let n_j be the number of essential interruptions (i.e., interruptions which left that program unrunnable) suffered by program u_j during the time period between t_j and T_j. Then the average system-wide PSI is given by

$$\Psi_s = \sum_{j=1}^{m} n_j/m. \tag{4}$$

Equation (4) is unsuitable for use in a live operating environment, because obtaining the required data would entail excessive overhead. Fortunately, it is possible to obtain a good approximation for Ψ_s using a minimum of data; in fact, all the required data were available on the CP-67 system at the Cambridge Scientific Center. The following system counters were used:

- P Total number of page exceptions
- V Total number of virtual I/Os
- W Total number of core-table pointer cycles
- M Cumulative MPL-elapsed time product, i.e., if $x(t)$ is the multiprogramming level at time t, then at time T, $M(T) = \int_0^T x(t)\,dt$.

Let $\Delta P, \Delta V, \cdots$, denote the differences between the counter values at the beginning and end of our time period, which will be assumed to be of length ΔT. The total number of essential interruptions suffered by all programs during that period is approximately $\Delta P + \Delta V$, and the average MPL is $\Delta M/\Delta T$, so that the average number of essential interruptions suffered by any one program is approximately $I = (\Delta P + \Delta V)\ \Delta T/\Delta M$. On the other hand, it was shown previously that an unreferenced page survives, on the average, 1.5 cycles. During the period ΔT there were $\Delta W/1.5$ such average survival periods. Hence, the average number of essential interruptions survived by unreferenced pages was $1.5\ I/\Delta W$. But this is, by definition, the system-wide PSI. Hence Ψ_s should approximately equal the value Ψ_s* obtained by evaluating

$$\Psi_s^* = \frac{1.5\ (\Delta P + \Delta V)\Delta T}{\Delta M \Delta W}. \tag{5}$$

One might ask how good an approximation to Ψ_s is Ψ_s*. In the course of the simulation runs described below, both Ψ_s and Ψ_s* were calculated at the end of each period of 30 seconds or 10 cycles, whichever came first. The two values agreed very well with each other in the region $0 < \Psi_s^* \leq 20$, which fortunately is the one of greatest interest. In a typical run containing 40 samples, the average values of Ψ_s* and Ψ_s were 12.61 and 12.67, respectively, the standard deviations were 4.09 and 4.16, and the correlation between them was 90.5 percent. It is concluded, then, that Ψ_s* is an excellent estimate for Ψ_s.

When Ψ_s exceeds 20, paging activity is so low that pages tend to remain resident until the program owning them becomes deactivated. Under these conditions Ψ_s no longer reflects the true survival capability of unreferenced pages. This accounts for the fact that Ψ_s tends to be much lower than Ψ_s* in this range.

6. Effect of paging on system performance

In order to devise a strategy for controlling system paging, it was necessary to determine what effect the various paging-related variables had on system performance. The following analysis was based on data collected under a real production load over a one-month period on the CP-67 system at the Cambridge Scientific Center. The system configuration contained 768 kbytes of core storage, with approximately 130 4096-byte page frames available for paging. Three IBM 2301 magnetic drums, mounted on two channels, were used as paging devices. The methods of data collection and analysis have been described elsewhere [5]. The data consisted of the values of various system counters recorded at approximately 100-second intervals (less frequently during slack usage periods). From these counter readings it was pos-

sible to calculate, for each observation period, the values of the following variables:

• *Performance variable*

1. Percent problem state time, i.e., the percentage of time during which the CPU was executing user programs

• *Key variables*

2. Average multiprogramming level
3. Ψ_S*
4. Average paging rate, page reads per second
5. Page steal ratio, i.e., fraction of pages read which replaced pages belonging to active users.

For the purposes of this study, percent problem state time was chosen as the primary performance criterion. The analysis proceeded with all observations being classified into groups according to the values of one of the key variables. The average percent problem state time was computed within each group of observations, and plotted against the values of the key variable (Figs. 2–4).

It was determined that performance is not sensitive to the MPL. This results from the fact that the CP-67 scheduler [6] attempts to control the MPL so as to optimize performance. The data do not suggest that maintaining a fixed upper limit on the MPL would be beneficial. On the other hand, Figs. 2-4 indicate that performance degrades when the paging rate exceeds 65 page reads per second, when the steal ratio exceeds 80 percent, or when Ψ_s* falls below 13. Still, it is remarkable that excellent performance is obtained at what might be thought of as very high paging rates and steal ratios. This is due, at least partly, to the simplicity and efficient coding of the CP-67 paging algorithms, which result in extremely low CPU overhead per paging operation.

7. Relation between PSI and program behavior

The effect of the PSI on the behavior of a specific program is readily derivable from the Ψ-realizations or from the paging characteristics. For instance, the average resident set sizes $A(\Psi)$ for various programs are plotted as functions of Ψ in Fig. 5. Most of these programs seem to follow similar behavior patterns: initially $A(\Psi)$ increases roughly in proportion to Ψ, but when Ψ reaches a critical value Ψ_c, somewhere between 7 and 10, the curve flattens out. Further increases in $A(\Psi)$ are proportional to Ψ^β, with β between 0 and 0.25. This suggests that by plotting the ratio $A(\Psi)/A(\Psi_c)$ vs Ψ/Ψ_c, one would obtain a generic representation (Fig. 6), valid for a wide range of programs.

Similarly, the paging rates $R(\Psi)$ of the various programs can be plotted vs Ψ (Fig. 7). Initially the paging rate decreases rapidly, in porportion to $\Psi^{-\gamma}$ (mostly $2 \leq$

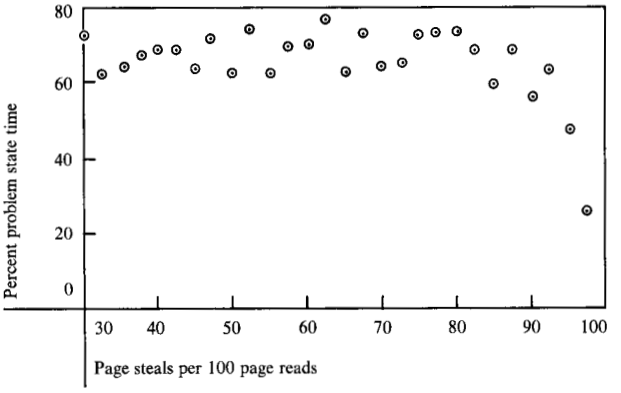


Figure 4 System performance vs steal ratio.

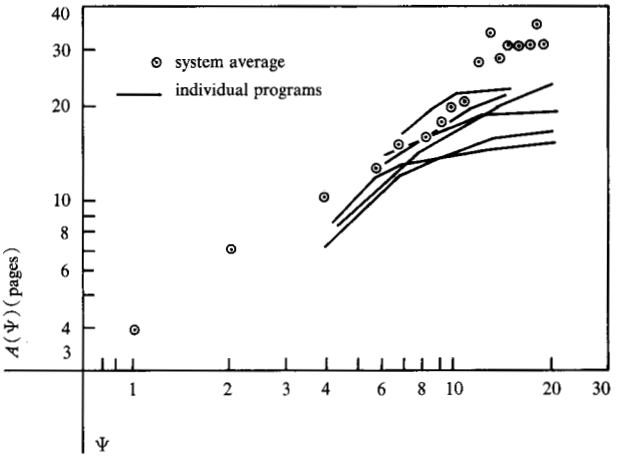
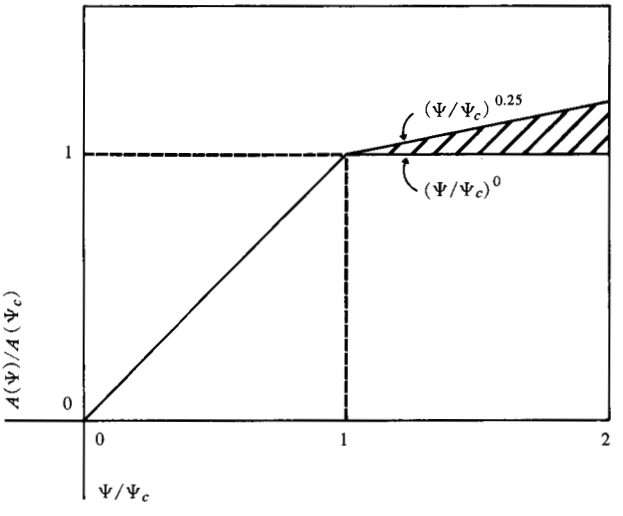


Figure 5 Program storage requirements.

Figure 6 Generic program storage requirements.



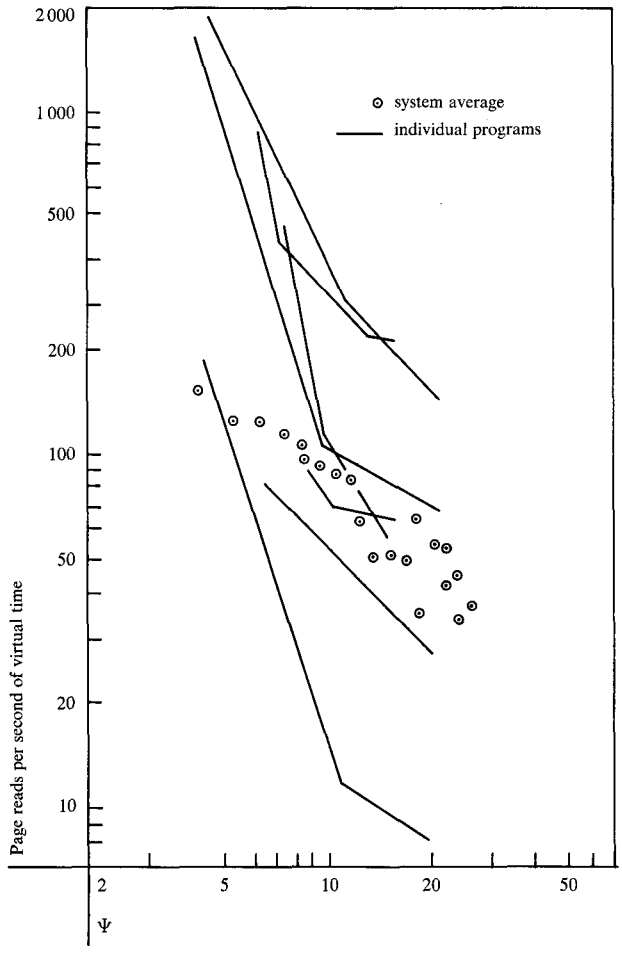


Figure 7 Program paging rates.

$\gamma \leq 4$), but again a break occurs at Ψ_c, and above that value the paging rate decreases only as Ψ^{-1}. The break, however, is absent in some cases, where $R(\Psi)$ is proportional to Ψ^{-1} over the entire measurement range.

These characterizations apply to specific programs whose execution traces were analyzed in detail. It is possible, however, to obtain a similar characterization for the "average" program actually running on the system in a production environment. Most of the required data for the period under study were available, as described in the previous section. The average resident set sizes could be estimated, for each observation period, by means of the formula

$$A^* = N/X, \tag{6}$$

where N is the total number of pageable pages available on the system and X is the average MPL. A plot of A^* vs Ψ_s^* has been superimposed on Fig. 5. It appears that the "average" program behavior is not dissimilar to that of the specific programs analyzed. The critical Ψ value seems to be higher (around 13, the value below which performance degrades, as shown in Fig. 2). However, this may be due to the fact that A^* is an overestimate when system activity is low, because active programs then occupy only a fraction of the N available page frames.

A plot of the overall system-paging rate vs Ψ_s^* (superimposed on Fig. 7) fails to show a breakpoint. A possible reason for that is suggested in Section 12.

8. Scheduling principles

Before applying the preceding results to the scheduling problem, we shall digress briefly to describe the principles by which the system scheduler carries out its functions. The primary function of the scheduler is to select the set of active programs, i.e., the set of users to be multiprogrammed. The dispatcher (not discussed further here), chooses which active program will actually run at any particular moment.

The scheduler performs its function in two steps: First, each eligible user is assigned a priority. Second, as many of the top-priority users as is deemed appropriate are admitted into the active set. The priority-assignment functions of the scheduler are matters of installation policy and will not be considered here. We are primarily interested here in determining how many users should be allowed in the active set, given that the users must be admitted in the predetermined order of their priorities.

It is generally recognized that a high MPL improves throughput by permitting a high degree of overlap in the utilization of various system resources. On the other hand, a high MPL may result in excessive paging. The scheduler must limit the MPL in such a way that main storage is not overcommitted. There are two complementary strategies employed to achieve this end: (1) *Feed-forward control*, in which the scheduler estimates each program's main storage requirements and admits programs only as long as their requirements can be met; and (2) *Feedback control*, in which the scheduler reduces the MPL if performance degrades due to excessive paging.

A balanced use of both control strategies appears desirable. In the succeeding sections we show how the PSI plays a crucial role in both feed-forward and feedback control.

9. Feed-forward control

Ideally, given all required data and unlimited computing resources, the scheduler would predict system performance for each permissible active set (in the case of a priority-ordered eligible list, the nth permissible set consists of the n top-priority programs). It would then

choose the set giving best performance (e.g., highest percent problem state). Performance prediction would proceed as follows:

1. Solve Eq. (1) for Ψ^*.
2. Predict paging and other I/O rates using Eq. (2) and its analogues.
3. Predict CPU utilization using, say, a cyclic queuing network model [7].

For obvious reasons this scheme is impractical. A more modest scheme is suggested by the results of Section 6:

1. Select a minimum acceptable value of the PSI, say Ψ_m.
2. Continue to admit programs to the active set as long as $\Sigma_i A_i(\Psi_m) \leq N$, the total number of available pages.

In this way, each program will be guaranteed enough pages for the whole system to run at an acceptable level of Ψ.

It remains now to predict the value $A(\Psi_m)$ for each program's next activation. This prediction is made in two steps:

1. Estimate $A(\Psi_m)$ for the program's previous activation. This is accomplished as follows: For each activation of a user, the system maintains a cumulative sum of the number of resident pages, the sum being incremented at each page exception. Let the value of this sum at the end of the activation period be S, and let Q be the number of page exceptions generated by the program during this activation. Then S/Q is the average number of resident pages (a correction for the initial page build-up period may be applied). Furthermore, let Ψ_s^* (Eq. 5) be calculated for the period in question. Then we take $S/Q = A(\Psi_s^*)$. Fig. 6 may now be used to estimate $A(\Psi_c)$:

   $$\text{If } \Psi_s^* < \Psi_c, \text{ then } A(\Psi_c) = (\Psi_c/\Psi_s^*)A(\Psi_s^*). \quad (7)$$

   $$\text{If } \Psi_s^* \geq \Psi_c, \text{ then } A(\Psi_c) = (\Psi_c/\Psi_s^*)^{\beta}A(\Psi_s^*), \quad (8)$$

   where β is some small exponent. The transformation from $A(\Psi_c)$ to $A(\Psi_m)$ now proceeds in reverse order. That is, in Eqs. (7) and (8) replace all occurrences of Ψ_s^* with Ψ_m, and solve for $A(\Psi_m)$. However, in the scheduler implementation that we have tested, we simply took $A(\Psi_c) = A(\Psi_m)$, which is quite reasonable if Ψ_m equals or slightly exceeds Ψ_c.
2. Estimate $A(\Psi_m)$ for the next activation. Some experiments on the prediction of working set sizes [8], and also analysis of successive activations of actual programs on a VM/370 system, have shown that the following technique produces predictions almost as good as those obtained by the best linear predictor that could have been constructed if the entire sequence of memory requirements were known in advance.

Let A_k be the value of $A(\Psi_m)$ computed for the kth activation of the user under consideration. Let $\overline{A_0}$ be some constant [the system's default guess for $A(\Psi_m)$]. Let $\overline{A_k}$ be an exponentially smoothed average of $A_1, A_2 \cdots, A_k$, computed by means of

$$\overline{A_k} = \alpha\overline{A_{k-1}} + (1-\alpha)\,A_k, \quad (9)$$

where α is a constant. Then the estimate for the next activation is

$$\overline{A_{k+1}}^* = \overline{A_k} + \rho(A_k - \overline{A_k}), \quad (10)$$

where ρ is a constant. Theoretically, ρ should be the lag-1 autocorrelation coefficient of the A_k sequence.

The CP-67 and VM/370 systems recognize two types of activations: interactive (Q1 stays) and non-interactive (Q2 stays). A program becomes a Q1 candidate after a console interaction has taken place. It receives a maximum of 0.25 seconds [9] CPU time during its Q1 stay. If not finished, it will become a Q2 candidate, and will receive up to 2.5 seconds [9] of CPU time during each Q2 stay. It is necessary to have separate values of $\overline{A_0}$, and to compute separate values of $\overline{A_k}$ and A_{k+1}^* for each type of activation. After each activation, the values of $\overline{A_k}$ and A_{k+1}^* are updated only for the type of activation just terminated. Conversely, when a program becomes a candidate for activation, only the A_{k+1}^* for the prospective activation type is used for testing storage availability.

10. Feedback control

Feed-forward control attempts to run the system at a Ψ_s^* level above some critical value. It is only natural that the scheduler should check periodically to determine whether this goal was achieved, and to take corrective action if it was not. Control actions may be based on other variables as well. For instance, Figs. 3 and 4 suggest that limits should be imposed on paging rate and steal ratio.

A typical set of control actions may be described as follows:

Strategy 1 (1) If at least one control variable falls outside the prescribed limit, reduce the maximum allowed MPL below the current level. Allow this limit to be reached by attrition, i.e., do not deactivate any programs immediately; (2) However, if all control variables fall outside their limits, remove some programs from the active set immediately; and (3) After all variables have returned to their permitted operating ranges, the limit imposed in (1) is gradually relaxed.

This set of rules is somewhat arbitrary and many others can be devised. The following were also tested in the simulation experiments described below:

Table 1 Principal Ψ-scheduler parameters.

Parameter	*Default value*	*Description*	*Used in equation*
Ψ_c	13 for 768-kbyte storage 11 for 512-kbyte storage	Critical Ψ value used to estimate storage requirements	7, 8
Ψ_m	13 for 768-kbyte storage 11 for 512-kbyte storage	Minimum allowed Ψ value	
P_M	65 page reads/second	Maximum allowed paging rate	
S_M	0.8	Maximum allowed steal ratio	
$\overline{A}_0^{(1)}$	5 pages	Initial estimated interactive storage requirement	9
$\overline{A}_0^{(2)}$	20 pages	Initial estimated noninteractive storage requirement	9
α	0.9	Exponential smoothing constant	9
ρ	0.5	Regression to mean constant	10
β	0	Exponent for $\Psi > \Psi_c$	8

Table 2 Simulation results: Comparison of CP-67 and Ψ-schedulers (all parameters at default values).

Workload	*Main storage size (kbytes)*	*Problem state time (percent)*	
		CP-67 scheduler	*Ψ-scheduler*
1	768	43.9	49.8
2	768	59.6	63.1
3	768	55.6	62.1
1	512	28.2	29.8
2	512	41.3	44.2
3	512	43.6	47.0

Strategy 2 Proceed as in strategy 1, except that in step (1), instead of limiting hte MPL explicitly, reduce the value of N used in testing whether $\Sigma A_i \leq N$.

Strategy 3 Proceed as in strategy 2, except that as soon as step (1) is invoked, immediately deactivate as many programs as necessary to meet the new storage restriction. Step (2) is now superfluous.

11. Simulation experiments

A scheduler (called the *Ψ-scheduler*) employing the principles of the preceding sections was implemented within a detailed trace-driven simulator of the CP-67 system [10]. The priority-setting and dispatching functions were left as in the CP-67 system [6]. The scheduler contains several parameters; the principal ones are listed in Table 1.

Each simulator run consisted of simulating 660 seconds of system operation with 40 logged-on users. The scenario of programs included executions of assemblies, compilations, object programs, APL functions, and file editing commands. Three distinct combinations of user scenarios made up the three workloads tested. There were no compute-bound programs in workload 1, there were some in workload 2, and more in workload 3. Two system configurations, differing in the amount of main storage available, were simulated. Percent problem state time was again taken as the primary performance measure, but other performance measures would have yielded similar conclusions. Experience has shown that differences of up to two percentage points between runs can be caused by such external factors as the user log-in order, or the initial placement of pages on the drums [11]. Hence, differences of less than this magnitude must be considered insignificant.

Several series of experiments were carried out. First, the performance of the Ψ-scheduler was compared to that of the CP-67 scheduler [6]. For these runs the default parameter values were used. The results appear in Table 2, and show the Ψ-scheduler to be superior in all cases. This is particularly significant, because the standard CP-67 scheduler had already proven its mettle in achieving high performance on real systems over long periods of time, as evidenced by the results shown in Figs. 2–4.

After establishing the attractiveness of the Ψ-scheduler, one would like to determine whether additional performance improvements can be obtained by tuning the parameter values to the particular workloads at hand. Whereas no systematic effort to locate optimum parameter values was made, a $3 \times 2 \times 2$ factorial experimental design was employed to determine the effects of some of the parameters. Two additional experiments (Nos. 13 and 14) were made to test intermediate values of Ψ_c. The results are displayed in Table 3, and show (even after formal statistical analysis) that no significant effects exist, even though it may be said that the performance attained with the original set of parameters could always be exceeded. If one had to make a choice at this

Table 3 Simulation results: Effect of Ψ-scheduler parameters (main storage = 512 kbytes; all other parameters at default values).

Experiment number	*Feedback strategy (see Section (10)*	$\Psi_c = \Psi_m$	β	*Probem state time (percent)* *Workload 1*	*Workload 2*	*Workload 3*	*Overall average*
1	1	11	0	29.8	44.2	47.0	40.3
2	1	11	0.25	32.7	44.0	46.6	41.1
3	1	10	0	31.1	46.6	47.5	41.7
4	1	10	0.25	32.3	46.1	47.0	41.8
5	2	11	0	32.7	44.2	48.7	41.9
6	2	11	0.25	32.2	45.0	49.0	42.1
7	2	10	0	32.5	44.4	48.1	41.7
8	2	10	0.25	31.2	45.8	48.4	41.8
9	3	11	0	32.9	42.1	49.0	41.3
10	3	11	0.25	31.8	43.7	50.6	42.0
11	3	10	0	31.5	43.3	50.9	41.2
12	3	10	0.25	33.5	45.7	49.5	42.9
13	2	10.5	0	33.4	44.3	49.0	42.2
14	2	10.5	0.25	31.4	45.9	48.3	41.9

point, one would select the parameter values for experiment 12. From an implementation viewpoint, however, the value $\beta = 0$ is preferable so one might choose the values from experiment 13.

A further series of experiments was carried out to determine whether feedback control on Ψ alone was sufficient, and how the algorithm performed under widely different values of Ψ_c and Ψ_m. These experiments constituted a 3×3 factorial design in which Ψ_c and Ψ_m were assigned all combinations of the values 5, 10, and 20, and all limits on paging rate and steal ratio were removed. The results (Table 4) show that control on Ψ alone was sufficient to maintain good performance, and that serious performance degradation occurred only when both Ψ_c and Ψ_m were set to extreme values. Best overall performance was obtained when both Ψ_c and Ψ_m had the "natural" value 10. By and large, however, either feed-forward control alone ($\Psi_c \geq 10$), or feedback control alone ($\Psi_m \geq 10$) were sufficient to maintain adequate performance.

Table 4 Simulation results: Ψ-scheduler control on Ψ alone (main storage = 512 kbytes; feedback strategy 3; $\beta = 0.25$; $P_M = 200$; $S_M = 1$; other parameters at default values).

Experiment number	Ψ_m	Ψ_c	*Probem state time (percent)* *Workload 1*	*Workload 2*	*Workload 3*	*Overall average*
1	5	5	17.9	39.0	41.6	32.8
2	5	10	31.5	46.1	47.9	41.8
3	5	20	33.4	45.9	49.3	42.9
4	10	5	30.4	45.0	45.0	40.1
5	10	10	33.3	46.9	49.4	43.2
6	10	20	32.0	44.1	47.6	41.2
7	20	5	32.4	47.1	47.2	42.2
8	20	10	32.1	45.1	50.7	42.6
9	20	20	28.5	42.3	47.6	39.5

12. Summary

If one views the value of the PSI as the prime determinant of system performance, then it becomes clear that the scheduler should function by maintaining the PSI at a desirable level. Hence, the feed-forward strategy aims to predict program storage requirements at that desired level, and the feedback strategy takes corrective action whenever the predictions have turned out to be drastically wrong.

Previous scheduling algorithms have generally attempted to control the paging rate directly. The CP-67 scheduler [6] attempted to estimate for each program separately the amount of storage it would require to page at what was considered a desirable rate. Then, if all active programs paged at that rate, the entire system would also page at that rate. Such a scheme, however, does not succeed: Unless controls are maintained on the amount of storage available to each active program, the total available space will split itself among the programs according to Eq. (1), and each program will assume its own paging rate at the resulting value of Ψ. On the other hand, controlling the amount of storage available to each program (as is done in so called working-set [12] scheduling algorithms) appears undesirable. In every case that we have examined, unrestricted sharing of main storage produced far fewer page exceptions than did running the same programs in fixed partitions. Furthermore, even if fewer exceptions were incurred by storage management schemes which impose such controls, they are likely to require more CPU overhead, and experience with CP-67 and VM/370 has shown that low paging overhead is

more important for good performance than low paging rate. There are, of course, special cases where such controls may be required, e.g., when one wishes to guarantee good response to special tasks.

Examination of Eq. (2) shows that overall system paging rate is determined primarily by slow-paging programs (large t_i). Fast-paging programs (small t_i) have relatively little effect. What happens, essentially, is that the fast-paging program doesn't get to run much, hence it can't steal too many pages from the other programs. Of course, if all active programs page fast, the entire system will also page fast. Figure 8 suggests that on our system there were usually enough slow-paging programs around to keep the overall paging rate from increasing at a superlinear rate, even when Ψ was below the critical level. This is a further reason for suggesting that one need not attempt to control the paging rate of individual programs (unless one desires to guarantee good response to these particular programs).

For the workloads considered in the simulation runs it appeared that no direct control of system paging rate was required. In practice, however, such control still appears desirable; otherwise one could not prevent the occasional occurrence of active sets that page excessively even at usually acceptable levels of Ψ. Unfortunately, the proper paging-rate control level (P_M) is likely to be quite configuration-dependent, whereas the critical Ψ values could be almost configuration-independent.

The preceding remarks raise the question of system tuning: How can one determine the best parameter values for a given installation? From the point of view of tuning ease, the Ψ-scheduler has two desirable properties: First, as demonstrated by the simulation runs, its performance is insensitive to fairly wide variations in the parameter values. Second, good initial values for many of the parameters are obtainable by simple analysis of system performance data: The values of Ψ_m, P_M, and S_M, can be read off plots such as Fig. 2-4; $\overline{A}_0^{(1)}$ and $\overline{A}_0^{(2)}$ can also be derived from usually available data; whereas Ψ_c, β, α, and ρ are probably sufficiently system-independent to make the values used here a good starting point at any installation. If further refinement is desired, experiments in which the parameters are varied automatically on-line can be designed and carried out as described in [13].

13. Acknowledgments

The author thanks C. Tillman for implementing the Ψ-scheduler on the CP-67 simulator and for many valuable suggestions regarding the design of the scheduler, P. Bryant and M. Schatzoff for their careful reading of the manuscript, and P. Callaway for supplying the VM/370 data.

References and note

1. Y. Bard, "Characterization of Program Paging in a Time-Sharing Environment," *IBM J. Res. Develop.* **17,** 387 (1973).
2. Y. Bard, "Prediction of System Paging Rates," *Proceedings of Computer Science and Statistics* (Seventh Annual Symposium on the Interface), Iowa State University, 1973, pp. 138-141.
3. *CP-67/CMS System Description Manual*, Form No. GH20-0802-2, IBM Data Processing Division, White Plains, NY, 1971.
4. *IBM Virtual Machine Facility/370*, Form No. GC20-1800 (Introduction), IBM Data Processing Division, White Plains, NY, 1972.
5. Y. Bard, "Performance Criteria and Measurement for a Time-Sharing System," *IBM Syst. J.* **10,** 193 (1971).
6. M. Schatzoff and L. H. Wheeler, "CP-67 Paging Priority Dispatcher," *Technical Report No. 320-2088*, IBM Scientific Center, Cambridge, MA, 1973.
7. J. Buzen, "Analysis of System Bottlenecks Using a Queuing Network Model," *Proceedings of ACM Workshop on System Performance Evaluation*, Harvard University, 1971, pp. 82-102.
8. P. Bryant, "Predicting Working Set Sizes," *IBM J. Res. Develop.* **19,** 221 (1975), this issue.
9. These figures apply to CP-67 running at the Cambridge Scientific Center. They may differ at other installations, particularly for VM/370 systems running on different-speed CPUs.
10. C. Boksenbaum, S. Greenberg, and C. Tillman, "Simulation of CP-67," *Technical Report No. G320-2093*, IBM Scientific Center, Cambridge, MA, 1973.
11. M. Schatzoff and P. Bryant, "Regression Methods in Performance Evaluation: Some Comments on the State of the Art," *Proceedings of Computer Science and Statistics* (Seventh Annual Symposium on the Interface), Iowa State University, 1973, pp. 48-57. In particular, see Table 5.
12. P. J. Denning, "The Working Set Model for Program Behavior," *Commun. ACM* **11,** 323 (1968).
13. Y. Bard, "Experimental Evaluation of System Performance," *IBM Syst. J.* **12,** 302 (1973).

Received June 20, 1974

The author is located at the IBM Data Processing Division Scientific Center, 545 Technology Square, Cambridge, Massachusetts 02139.