Y. Bard

Characterization of Program Paging in a Time-sharing Environment

Abstract: This paper describes a method for predicting the paging behavior of a program in a virtual memory multiprogramming environment. The effect of overall system activity on the program is summarized in one parameter, the page survival index. The model correlates well with observations taken on programs running under CP-67. The model can be used for paging load prediction, simulator input verification, and evaluation of program rearrangement and sharing.

1. Introduction

With the advent of virtual memory it has become important to characterize the paging behavior of programs. One wishes to know what computer resources are necessary to make a program run efficiently, or conversely, how a program will run with given resources. The two established methods of characterizing program behavior answer these questions under certain conditions. If a program is to be run in a memory partition of fixed size, then the so-called "parachor curve" [1] predicts the number of page exceptions that will occur. If the program is to be dispatched in fixed time slices, then knowledge of its working sets [2] enables one to assign the proper memory size for each time slice. These approaches, however, cannot deal adequately with a program running in a multiprogramming environment, where neither the memory allotted nor the time that the program is allowed to run is fixed. It seems intuitively clear that the most efficient use of system resources can be achieved only if their allocation to the various users is allowed to vary dynamically in response to the users' needs, rather than according to some predetermined scheme. Indeed, the trend is toward designing time-sharing and other multiprogramming systems in this way [3].

In section 2, the behavior of a program running in this kind of system is described. The key difference from its behavior in a system with a fixed memory size is that the other programs running affect this program's behavior, particularly the typical length of time that an unreferenced page remains in main storage. A formal measure of this effect, the page survival index, is defined in section 3, and in section 4, it is shown how, for a given value of the page survival index, the execution trace of an individual program may be used to derive approximate measures of the program's performance in the "true" environment. These measures, which we call Ψ-realizations, are analogous to parachor curves and working sets for characterizing behavior of single programs in a uniprogrammed environment.

The results of some experiments on Control Program-67/Cambridge Monitor System (CP-67/CMS) to verify the resulting model are described in section 5. Comparisons of these results with the parachor curve approach are given in section 6. In section 7, extensions of the ideas of sections 2 and 3 in a CP-67 environment are given. Finally some applications of the model are described in section 8.

2. Description of program execution

We assume that a program (designated X in the sequel) is being executed under the control of a multiprogramming system, on a computer consisting of a single central processing unit, a main storage unit divided into fixed-size page frames, and suitable I/O equipment.

At any given time, a certain number of the program's pages reside in main storage. We refer to these pages as the *resident set*. The program is executed until interrupted by the occurrence of some event, such as one of the following:

1. The program references a page not currently in the resident set (*page exception*).
2. The program enters the (virtual) wait state, e.g., to await the completion of an I/O operation.
3. A timer interruption indicates end of time slice.

When the interruption occurs, the operating system gains control, initiates required actions (e.g., bringing in the page that caused the exception), and then dispatches some other program. The period from the time a program is dispatched until it suffers an interruption is called an *execution interval.*

When execution of program X is resumed at some later time, its resident set may have changed because:

1. The interruption was caused by a page exception, and the page causing the exception will have been brought in and added to the resident set.
2. Some pages previously in the resident set may have been removed by the operating system to make room for pages required both by X and by other programs.

The principal aim of this paper is to show how the trace of a program being executed *alone* can be used to predict approximately the sequence of execution intervals and resident sets that will occur when the program is executed together with a set of unknown programs in a demand-paging multiprogrammed environment. The effect of this environment is summarized in a single parameter, as described in the next section.

3. Page survival index

We assume that the operating system uses an LRU- like page replacement algorithm. This means that when a page demanded by a program is brought into main storage, it is likely to replace a page that has not been referenced for a relatively long period of time. Such algorithms are used, for instance, in CP-67 [4], VM/370 [5], MTS [6], and Multics [7].

A program can lose pages only while it is in the interrupted state. Therefore, the number of interruptions suffered by the program is a better indicator than elapsed time for the purpose of ascertaining which pages have been lost. With a strict LRU replacement algorithm, pages not referenced for a time equivalent to n interruptions will be lost before pages that have been referenced in a time equivalent to fewer than n interruptions. In the approximations to LRU that have been implemented on real systems, the clause "will be lost" should be replaced by "are more likely to be lost." However, we shall neglect this difference in the model.

Suppose the system paging rate is low. Under this condition, unreferenced pages can survive a relatively large number of interruptions. On the other hand, if the demand for pages is large, e.g., because the multiprogramming level is high, then unreferenced pages will tend to be lost after relatively few interruptions. The effect of overall system activity on individual program paging behavior can be summarized by means of one parameter, called the *page survival index* (PSI), which is defined as the number of interruptions that an unreferenced page can survive. Low PSI values are associated with high system activity levels, and vice versa. We shall use the letter Ψ to designate specific values of the PSI.

A specific interpretation of the PSI for the CP-67 system is given in a subsequent section. First, however, we show how program behavior can be characterized in a deterministic way, assuming that the value of the PSI prevailing in the system at the time of execution is a known constant.

4. Execution trace analysis

Suppose a complete execution trace is available for program X. The trace contains a sequential list of all instructions executed, each with a list of pages referenced by that instruction. Such a trace tells us directly how the program is executed alone in unlimited storage. Further analysis is required to show how the program is executed in the multiprogrammed environment. At the most detailed level, the desired characterization consists of a list of all execution intervals of the program. For each interval, we require the length of the interval (CPU time or number of instructions), the resident set, and the type of interruption that terminated the interval. Once these data are obtained, more concise characterizations can be computed. For instance, one may wish to determine the joint distribution of execution interval length and resident set size. Or, one may be content with simple summary statistics, such as average resident set size and number of page exceptions.

For a given value of the PSI, the detailed characterization, which we shall call the Ψ-*realization* of program X, may be obtained from the execution trace in the following manner:

Define the following counters:

T: total program execution time,
t: time since start of current execution interval,
m: instruction number,
k: interruption number.

For each page p referenced by the program we shall maintain a counter $L(p)$ containing the time that the page was last referenced.

For each interruption k we shall define values $S(k)$ containing the time of occurrence, and $U(k)$ containing a code defining the type of interruption. Assume $S(k) = 0$ for all $k < 1$. With each interruption k we shall also associate a vector P_k containing a list of the pages resident at the time of the interruption.

Let Ψ be the prevailing value of the PSI.

The algorithm proceeds as follows:

1. *Initialization*: Define the resident set to be the empty set, and let $T = 0$, $m = 1$, $k = 0$, and $L(p) = -1$ for all p.

2. *Execution interval initialization*: Let $t = 0$.
3. *Instruction initialization*: Check whether the mth instruction references any page not in the resident set. If it does, proceed to step 5).
4. *Instruction execution*: Increase T and t by the execution time of the mth instruction. Set $L(p) = T$ for each page referenced by the instruction. Increase m by 1. If the instruction does not cause the program to enter the wait state, and if t does not exceed the time slice length, return to step 3).
5. *Interruption*: Increase k by 1. Set $S(k) = T$ and store the interruption type code in $U(k)$. Define the vector P_k to consist of the current resident set page identifier.
6. *Resident set adjustment*: Remove from the resident set all pages for which $L(p) < S(k - \Psi)$. This simulates the loss of pages that have not been referenced during the last Ψ execution intervals. If the interruption is a page exception (a transfer from step 3) to 5)), add the offending page to the resident set. Note: If the instruction references several pages not in the resident set, only one is added at a time.
7. Return to step 2).

It is evident that upon completion of the trace, one has complete information on the lengths of the execution intervals, on the associated resident sets, and on the nature of the interruptions that occur, at the given value of the PSI. For instance the length of the kth execution interval is given by $T_{S(k+1)} - T_{S(k)}$, and the resident set size at the nth interruption equals the dimension of P_k. By judicious programming of the algorithm, it is possible to obtain realizations for several values of the PSI simultaneously during one pass through the trace. Once these realizations are obtained, one easily computes the summary statistics required for the various applications described below.

5. Model verification

The model was verified by comparing its predictions to data obtained from actual runs of various programs under the CP-67 system. This system maintains various counters that register the activities of each user on the system. The two counters relevant to the present study are the following:

1. N: number of page exceptions,
2. S: cumulative sum of resident set sizes, the sum being incremented at each page exception.

By forming the ratio $A = S/N$, one obtains the average resident set size (average number of pages in main storage), the average being over the instants at which page exceptions occur. This average is a measure of the cost to the system of the storage occupied by the program at the time that its execution is blocked by a page exception [3].

The values of N and A can be easily computed from the information recorded during generation of the realizations for various values of the PSI, as indicated in the preceding section. The resulting values are plotted in Figs. 1–4 for various programs whose traces were available. We refer to these plots as the *paging characteristics* of the programs. Also plotted are the values of N and A for replicate runs of the same programs under uncontrolled load conditions on CP-67 in an actual operating environment. Since the activity level of the system fluctuates rapidly, one cannot expect a single value of the PSI to prevail throughout a run. Hence, the observed points generally do not correspond to any specific predicted points, and only approximate agreement can be expected, with occasional wide departures. Some additional inaccuracies of the model are the following:

1. Interruptions of various types tend to differ in length. A page is less likely to survive the long wait for a console input than the short wait for a disk I/O.
2. Some interruptions may be conditioned on outside events. For instance, a program attempting to initiate an I/O operation will suffer an interruption if the device is busy serving another program.
3. Some interruptions may originate externally to the program under study, e.g., the preemption of the CPU by a program of higher priority.

Nevertheless, the agreement between the observed data and the locus of the predicted points is quite good.

6. Static vs dynamic storage requirements

The paging characteristics of Figs. 1–4 are analogous to the usual "parachor curves," where the number of page exceptions is plotted against the size of a fixed storage partition allotted to the program. These latter curves are also plotted on Figs. 1–4. They were obtained by running the programs alone under CP-67 in fixed storage partitions of various sizes and counting the number of page exceptions generated. It is clear from the figures that by allowing a program's resident set to expand and contract according to its needs one considerably reduces its *average* storage requirements over what would be necessary if the program were forced to be executed in a fixed-size partition. It is worth noting that the observed points are almost always much closer to the paging characteristics than to the fixed memory parachor curves.

7. Interpretation of the PSI in a CP-67 like system

In order to explain how the PSI manifests itself in a CP-67-like system, it is necessary to review the CP-67 Release 3.1 page replacement algorithm, i.e., the method

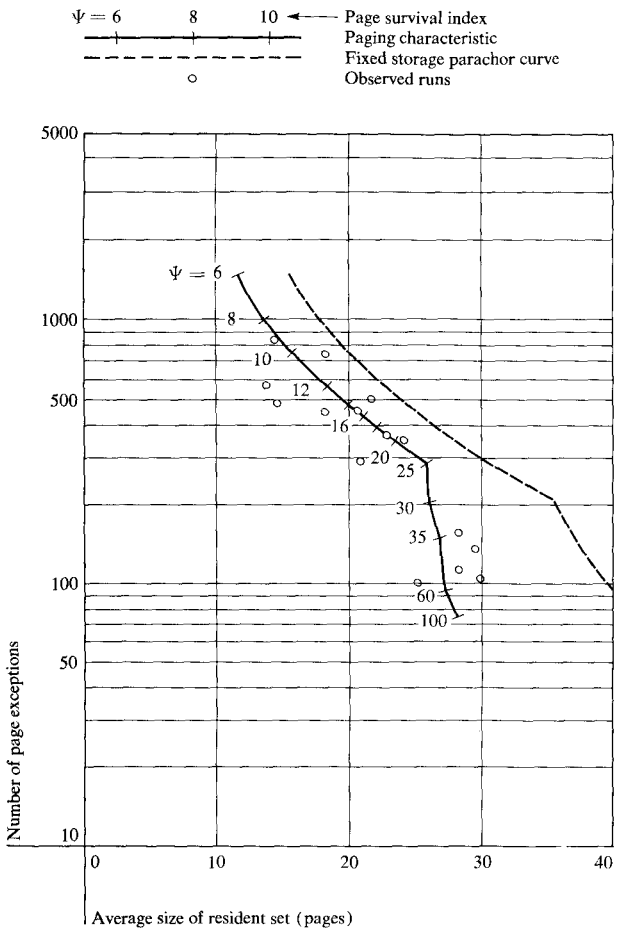

Figure 1 Assembly of 115-card deck.

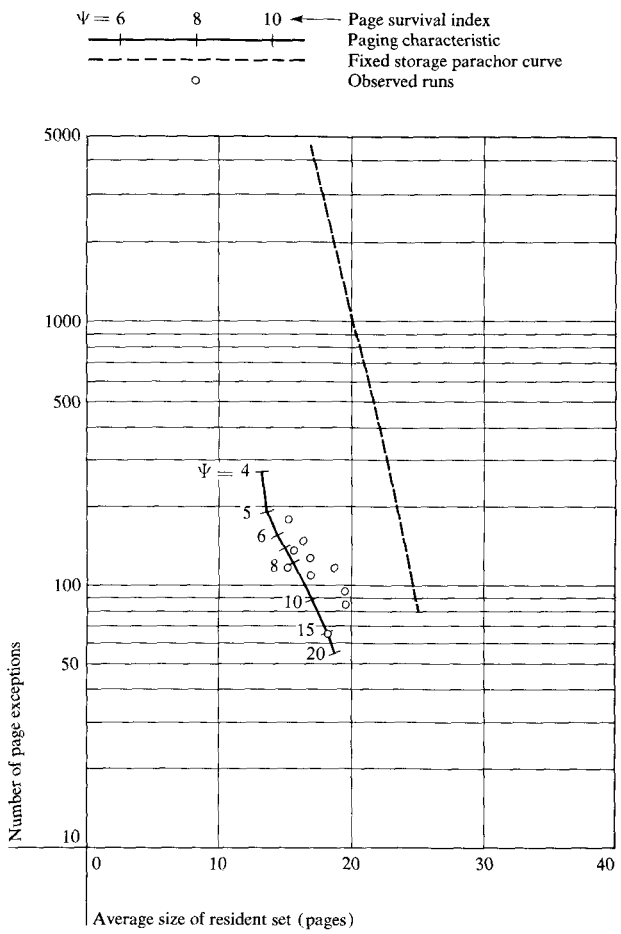

Figure 2 FORTRAN compilation of 49-card deck.

for choosing which page should be removed from main storage to make room for a new page to be brought in from auxiliary storage. We point out first that CP-67 maintains a table of all pages in main storage, a reference bit associated with each page, and a pointer that cycles around the 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 algorithm may now be described as follows:

Select the next page pointed to in the table 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.

We refer to the time that it takes the pointer to move through the entire table as a *cycle*.

A page that has survived the passage of the pointer has its reference bit off. The page will be lost if, and only if, it remains unreferenced until the next passage of the pointer. Any page that is lost has, therefore, gone unreferenced for at least one cycle. On the other hand, it is clear that a page cannot survive unreferenced for two full cycles. Suppose program X is dispatched $n(X)$ times during one cycle. Thus $n(X)$ is also the number of interruptions that the program has suffered, and it follows that:

$$n(X) \leq \Psi(X) < 2n(X),$$

where we have used the notation $\Psi(X)$ to refer to the value of the PSI that holds for this particular program. Assuming that the last reference to a page, which is known not to have been referenced for at least $n(X)$ intervals and at most $2n(X) - 1$ intervals, is distributed uniformly over this range, we deduce

$$E\Psi(X) = 3/2\ n(X),$$

where E denotes the expected value. Thus, in reality the value of $\Psi(X)$ would vary randomly over a certain

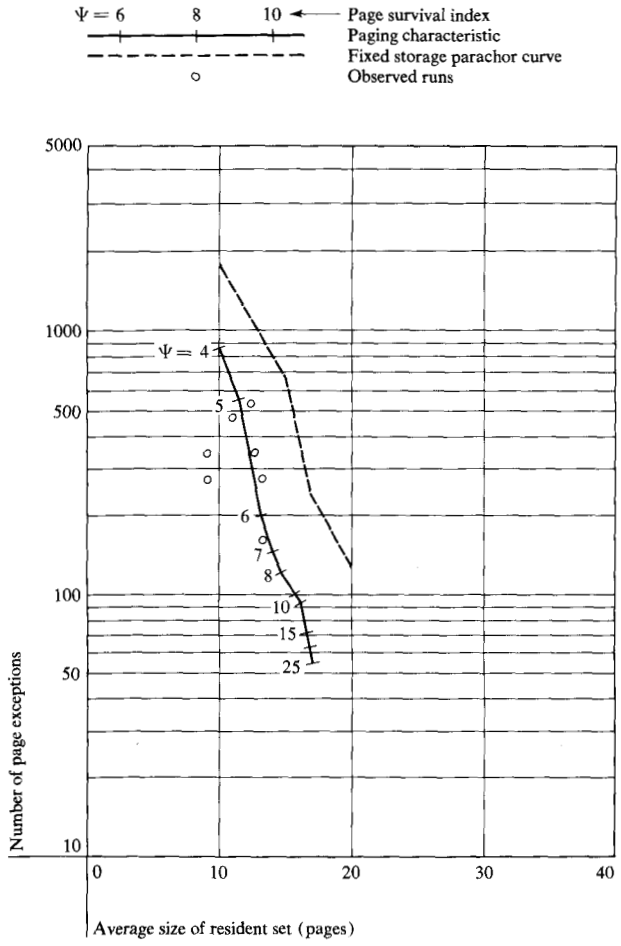

Figure 3 Execution of FORTRAN program.

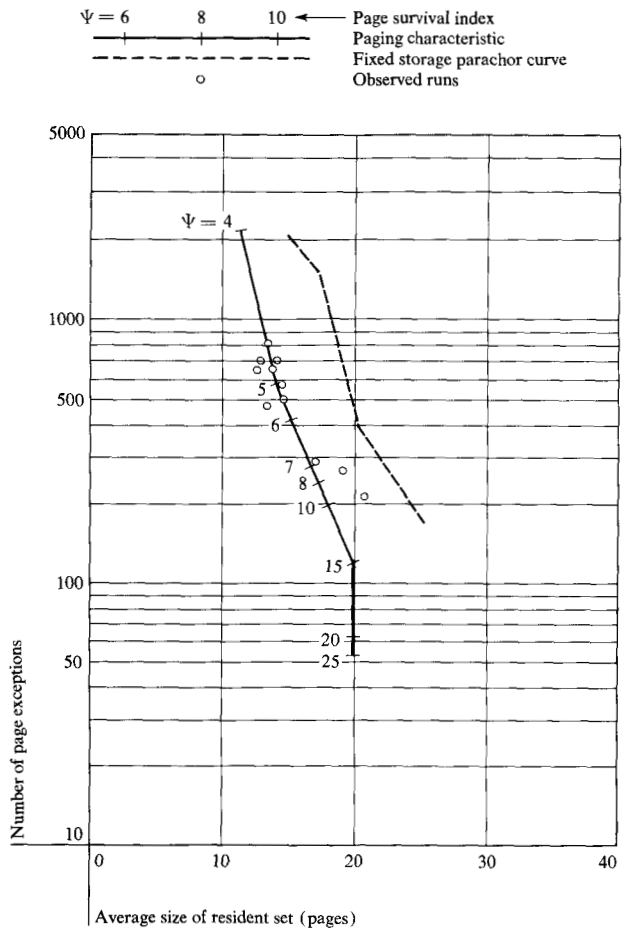

Figure 4 Execution of PL/I program.

range, whereas in our characterization we assumed it to be fixed.

Suppose that K programs are being multiprogrammed together, and suppose all these programs are dispatched at the same frequency, e.g., as in a round robin schedule. Since $n(X)$ will be the same for all programs, they will also have the same expected value of $\Psi(X)$, and we may henceforth drop the argument (X). Suppose further that all programs are running under nearly steady-state conditions, with no marked changes in the sizes of their resident sets.

Let $A(i, \Psi)$ be the average resident set of the ith program, given that the PSI is Ψ. If M is the total number of page frames available in main storage, then clearly

$$M \geq \sum_{i=1}^{K} A(i, \Psi).$$

Since A is generally a monotonic increasing function of Ψ (see Figs. 1–4), and since active programs tend to increase their resident sets at the expense of inactive programs, it follows that under steady-state conditions the maximum value of Ψ for which the above inequality holds will prevail, i.e.:

$$\Psi^* = \max \{\Psi | M \geq \sum_{i=1}^{K} A(i, \Psi)\},$$

where Ψ^* denotes the actually prevailing value of Ψ.

Let $R(i, \Psi)$ be the page exception rate (per second of execution time) for the ith program, given that the PSI is Ψ. Furthermore, let $t(i,\Psi)$ be the average length of the execution interval. Then the overall page exception rate (per second of virtual CPU time) is:

$$R^* = \frac{\sum_i R(i, \Psi^*) t(i, \Psi^*)}{\sum_i t(i, \Psi^*)}.$$

Let $f(i, \Psi)$ be the fraction of execution intervals terminating in a page exception. Since each program has, on the average, $n^* = 2/3\Psi^*$ execution intervals per cycle, it

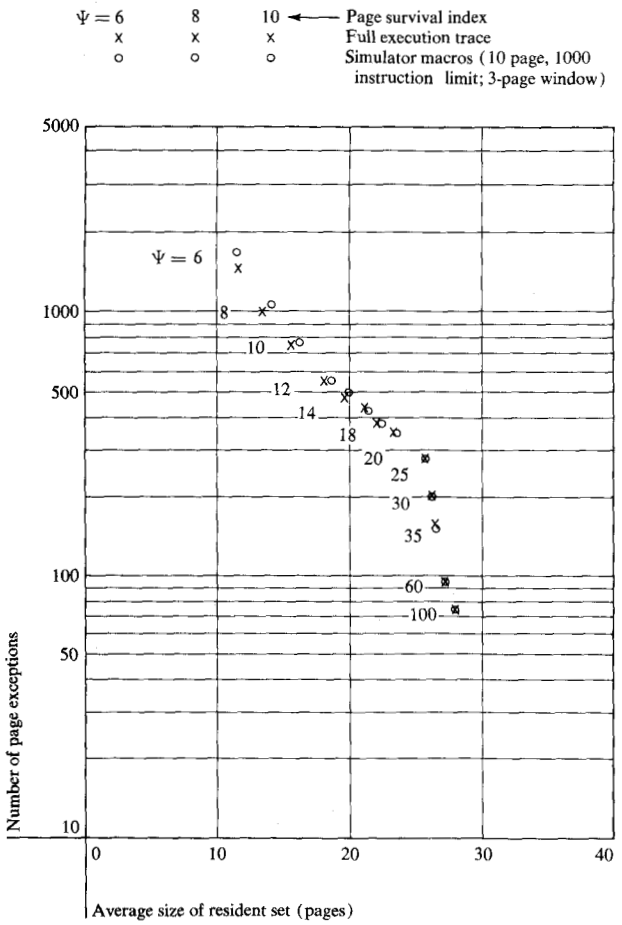

Figure 5 Use of paging characteristic in evaluation of trace compression technique.

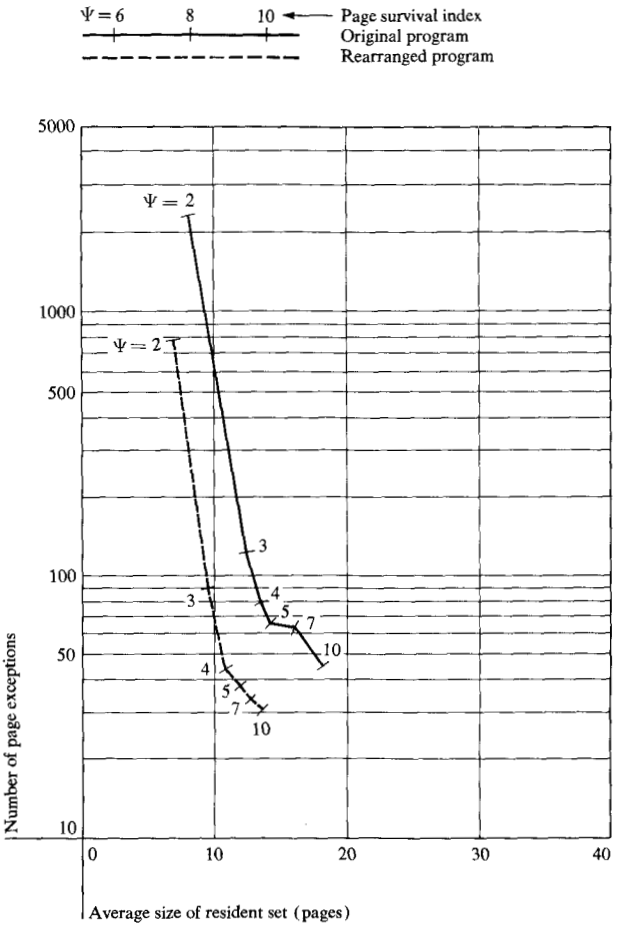

Figure 6 Use of paging characteristic in evaluation of program rearrangement.

follows that there are $F^* = 2/3\Psi^* \sum_i f(i, \Psi^*)$ page exceptions per cycle. Hence the duration of a cycle is F^*/R^*, and the pointer advances, on the average, M/F^* pages per page exception.

The functions $A(i, \Psi)$, $R(i, \Psi)$, $t(i, \Psi)$, and $f(i, \Psi)$ are easily calculated from the realizations described in Section 4, and all other quantities of interest can be computed from the above formulas. I/O rates (other than paging) can be computed similarly to the paging rate, and these rates can be used to predict CPU utilization and other throughput measures by means of suitable queuing models [8].

The assumptions of steady state and equal dispatching frequency are rarely met in practice, and the above equations were derived only to give the reader a feeling for the way in which the PSI is related to events occurring in the system. A more realistic model for predicting system behavior under a given set of programs running together is being developed.

8. Applications

The principal potential application of the model is in attempting to predict system performance that will result when a given set of programs is run together. A simplified approach to this problem has already been presented in the preceding sections.

An application for which the model has already been used is in verifying compacted descriptions of program execution traces to be used as inputs to a system simulator [9]. In this compaction, many machine instructions are aggregated into a single "macro instruction." A description of the macro instruction consists of the number of machine instructions included, and the set of pages referenced. Special macros exist for I/O instructions and other exceptional cases. Macro sizes are limited by the number of pages and/or number of machine instructions allowed. When fed to the simulator, a special "window" algorithm [9] is used to break up the macro into a simulated page reference string. Due to the manner in which

the macros are generated (fixed limit on pages referenced per macro) there is generally excellent agreement between the "parachor curves" of the simulated and real programs, at least for fixed memory sizes above the macro limit. Therefore, comparison of the Ψ-realization statistics provides a much more critical test of both the compaction and window algorithms. An example of such a comparison is given in Fig. 5. Incidentally, the process of generating the Ψ-realizations provides an excellent method for determining macro instruction boundaries. One could in fact argue that the macro instructions should correspond precisely to the execution intervals in the Ψ_0-realization of the program, where Ψ_0 is the greatest lower bound on the values of the PSI observed in the real system. It can be shown that for $\Psi \geq \Psi_0$, Ψ-realizations generated from such a compacted trace are exactly the same as those obtained from the full instruction trace, and the set of interruptions in the Ψ-realization is a subset of the interruptions in the Ψ_0-realization, provided time slice ends are ignored.

It is usually possible to write a program in many different, though functionally equivalent, ways. Our characterization could be used to determine which one of several implementations makes the least demands on the system's paging resources, i.e., by having the lowest paging characteristic. Hatfield and Gerald [10] have developed methods for rearranging the modules of a program so as to reduce paging. The effect of such a rearrangement is shown in Fig. 6, showing that the desired objective was, indeed, achieved: at each value of the PSI, there are savings both in average storage required and number of page exceptions.

From each realization of program X we can compute the fraction of execution time during which any specific page is resident, and also the number of exceptions caused by that page. As shown by Wahi [11], this information can be used to determine which pages of a popular procedure should be shared among users and/or locked permanently in main storage.

9. Conclusion

We have presented here a characterization of a program's behavior in a multiprogramming virtual memory system. The characterization is based on the concept of the resident set, which may be regarded as an adaption of the working set [2] concept. The latter captures the pages *referenced* during a *predetermined* time interval. The former captures the pages likely to be *resident* in main storage during an organically determined *execution* interval. In fact, the resident set at the kth interruption is precisely the working set for the period spanned by the preceding Ψ execution intervals. The page survival index (PSI) characterizes the interaction between the environment (the operating system and all the programs being executed) and the specific program under investigation. The Ψ-realizations of a program's execution are intended to be reproducible approximations to actual, nonreproducible executions of the program on the real system. We feel that statistics (such as the paging characteristic) generated from the sequence of execution intervals and resident sets predicted for various values of the PSI are more relevant for studies of program behavior and system design and evaluation, than those based on the more traditional concepts of fixed-time working set and fixed-storage parachor curve.

Acknowledgment

The author wishes to thank his colleague P. Bryant and the referees for their helpful comments and suggestions.

References

1. B. S. Brawn and F. G. Gustavson, "Program Behavior in a Paging Environment," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33,** 1019–1032 (1968).
2. P. J. Denning, "The Working Set Model for Program Behavior," *Communications of the ACM* **11,** 323 (1968).
3. M. Schatzoff and L. H. Wheeler, *CP-67 Paging Priority Dispatcher, IBM Cambridge Scientific Center Technical Report No. 320-2088*, Cambridge (1973).
4. *CP-67/CMS System Description Manual*, Form No. GH20-0802-2, IBM Data Processing Division, White Plains, NY (1971).
5. *IBM Virtual Machine Facility/370, Introduction*, Form No. GC20-1800, IBM Data Processing Division, White Plains, N.Y. (1972).
6. M. T. Alexander, "Time-Sharing Supervisor Programs," in Conference on Advanced Topics in Systems Programming (1971).
7. E. I. Organick, *The Multics System*, MIT Press, Cambridge, Mass. 1972.
8. 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.
9. C. Boksenbaum, S. Greenberg, and C. Tillman, "Simulation of CP-67," *IBM Cambridge Scientific Center Technical Report No. 2093*, Cambridge (1973).
10. D. J. Hatfield and J. Gerald, "Program restructuring for virtual memory," *IBM Systems Journal* **10,** 168 (1971).
11. P. N. Wahi, "On Sharing of Pages in CP-67," *Proceedings of ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, Harvard University, 1973, pp. 127–149.

Received December 7, 1972
Revised April 30, 1973

The author is located at the IBM DPD Scientific Center, Technology Square, Cambridge, Mass.