A STUDY OF THE EFFECT OF USER PROGRAM OPTIMIZATION IN A PAGING SYSTEM

Lesin W. Comeau
International Business Machines Corporation
Cambridge, Massachusetts

Summary

Much attention has been directed to paging algorithms and little to the role of the user in this environment. This paper describes an experiment which is an attempt to determine the significance of efforts by the user to improve the paging characteristics of his program.

The problem of throughput in a computing system is primarily one of balancing the flow of data and programs through a hierarchy of storages. The problem is considered solved when for every available processor cycle there is a matching demand for that cycle in the primary (execution) store. Since programs and their data usually originate in a location other than the execution store, there is a delay associated with the movement of data and programs to the primary store. The delay has two components, the operational speed (data transfer time) and the positioning, or access time, of the secondary storage device. Since the access time usually exceeds the data transfer time by an order of magnitude, the problem of transferring information to the primary store has been named the "access gap" problem.

Previous software solutions to the access gap problem divided the problem into two areas of responsibility. First, the problem programmer was responsible for buffering and blocking his data such that high CPU utilization was realized. Subsequent studies showed that this alone would not result in a significant overall increase in system throughput. As technology improved and CPU's and execution store cycles were speeded up, it became apparent that job set-up and initialization was responsible for low throughput performance. A second solution to the "access gap" problem, multiprogramming, was therefore implemented. In this approach, the system programmer is given the responsibility of buffering programs in much the same way the user (problem programmer) buffers data.

It is evident that the critical parameter in this multiprogramming environment is that of block size, since this will determine the number of buffers which can be created from available memory, which is the same as the level of multiprogramming present (i.e. the number of programs which can simultaneously reside in primary storage). The usual blocking factor in a multiprogramming environment is 'one'; that is, the program is treated as one block and a core buffer space is reserved for this amount. In much the same manner that the problem programmer finds when he blocks data this way, we find that programs which result in large block size usually have low core utilization with a resultant reduction in throughput. The low core utilization comes about in two ways. First, a portion of the program is normally never executed, and hence its primary store occupancy is not required. Second, sections of the program are linear in execution with respect to time, and once they have been used, they are not required again; but in systems in which programs are blocked in one unit this unnecessary code must remain in the primary store.

A method of artificial blocking has been devised in an attempt to solve this blocking factor problem incurred with programs. This technique, paging, first used to extend the address space of a small physical core[1], is now used to reduce the physical core requirements of programs written in a very large address space. The alternative to this approach is for the programmer to utilize only a small portion of the address space available (with resultant increase in the core available for multiprogramming) and to substitute an overlay structure for paging. This is distasteful for two reasons. First, it requires a pre-planning of routines -- which to overlay and which to leave resident. Preplanning in large programming systems is usually inadequate. Second, if only a small part of the available address space is utilized and there are no jobs to be multiprogrammed, then the remaining physical core is not utilized properly. Solution time for the problem is thereby needlessly increased.

There is in a paging system, however, one gross source of error. It is found in the very premise which makes paging in a large address space attractive in the first place, namely, the block size. Ideally, a multiprogrammed paging system attempts to pick a page size which closely emulates a natural grouping of instructions. Since the natural grouping is usually found to be quite small [2], a compromise page or block size must be derived, as a solution; otherwise the mapping hardware or the storage requirements for mapping tables involve considerable expense.

Unfortunately, those who emphasize paging as a solution to the multiprogramming problem fail to define the responsibility of the user in this environment.

It has been stated that the programmer can do nothing easily to improve the situation. [3] It is the contention of the author that this is not so and that, in fact, it is the responsibility of a programmer who executes in an algorithmically managed storage to adjust his programs to the page size if they are to be run frequently.

To prove this contention, the Cambridge Monitor System (CMS) [4] which was written to take no advantage of a paging environment, was picked for experimentation because its authors were resident and because it was the conversational monitor which would receive highest use on our system. We first asked one of the knowledgeable systems programmers to arrange the nucleus such that the resulting load list would place routines dependent on each other in the same 4k byte block, the page size of the 'virtual machine' system of the Cambridge Scientific Center [5,6]. The result of this, the aligned deck, is shown in Figure 1. Subsequently a data reduction program PSTA (Appendix A) was written which produces a map of core (see Fig. 2) as paging demands are incurred in program execution. In order to produce these demands, the amount of physical core available to the program was restricted to 64K (16 pages).

A second CMS programmer was given the PSTA output and told to structure his load list (i.e., to rearrange the load sequence of subroutines) so as to reduce the total page requirement. In three passes of data reduction, (PSTA output) a 50% reduction in the number of page transfers was achieved in this JAS arrangement from the aligned version (Fig. 1).

The experiment involved the assembly of a single deck in all runs. Only the nucleus which consisted of 64 movable subroutines, was resequenced. The assembler, which runs in conjunction with the nucleus, was not changed.

The total machine time involved was approximately 20 minutes, and the number of man hours totaled approximately 24, for the JAS deck. The aligned deck required only .5 man hours to arrange.

Since there had been little work in structuring programs to run in a paging environment, this effort was considered worthy of further investigation, namely, into the effect of applying no effort to achieve optimization. Since there were 64! possible arrangements with CMS nucleus, it is impossible to determine, if a deck was arranged in a random manner, the probability of doing better than the JAS arrangement; the following two experiments did show, however, that it could be significantly worse. The deck was sequenced in a random manner, and in an alphabetic fashion (which is a common arrangement that facilitates deck replacement while debugging); the results, as compared to the easily achieved improvement are rather significant (Fig. 1).

To restate the conditions of the experiment shown in Figure 1:

Alphabetic:	the decks were arranged in the alphabetic sequences of the names punched in columns 73-80.
Random:	decks were given sequential numbers from the JAS arrangement, then resequenced by a random number table.
Aligned:	the programmer rearranged his decks based on a knowledge of 4K byte pages.
JAS:	programmer, given a map of paging actions generated while his job was in execution, resequenced his load list for an improved paging performance.

The same experiment, using a Fortran job, was run to measure the difference between alphabetic and JAS.

Although the difference of 3.1 in this case is less significant, it must be noted that a level of optimization which was achieved in one environment (i.e., the assembly) had a beneficial effect in the other. This is what one would expect since the section of code optimized consisted of the interrupt and data management routines used by both the assembler and Fortran.

In view of such significant results, it must be emphasized that no code has been rewritten; only the relative order of the subroutines within a deck has been changed.

It is, therefore, the author's opinion that the user optimization is not only easily achieved, but absolutely necessary for frequent operation in a paging environment. Effort is now being directed to constructing an algorithm and program to perform the load list restructuring mechanically.

The author is most grateful to the CMS people who participated in this experiment, especially to Mr. J. B. Harmon, the project manager, who provided the able assistance of Mr. J. A. Seymour, who structured the JAS deck.

References

1. T. Kilburn, D. B. G. Edwards, M. J. Lenigan, and F. H. Sumner, "One Level Storage System" - IRE Transactions on Electronic Computers, Vol. EC-12, PP. 223-235, April, 1962.

2. O'Neill, R. W. - "Experience Using A Timesharing Multiprogramming System with Dynamic Address Relocation Hardware". AFIPS Conference Proceedings, Vol. 30, PP. 611-621, April, 1957.

3. G. H. Fine, C. W. Jackson, and P. V. McIssac, "Dynamic Program Behavior Under Paging", System Development Corporation SP-2397, June, 1966.

4. Cambridge Monitor System - A user's manual available from the Cambridge Scientific Center.

5. A. B. Lindquist, R. R. Seeber, and L. W. Comeau - "A Time-Sharing System Using an Associative Memory", Proceedings of the IEE, Special Issue on Computers, Vol. 54, No. 12, PP. 1774-1779, December, 1966.

6. R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy - "A Virtual Machine System for the 360/40", Cambridge Scientific Center Report 36.010, May, 1966.

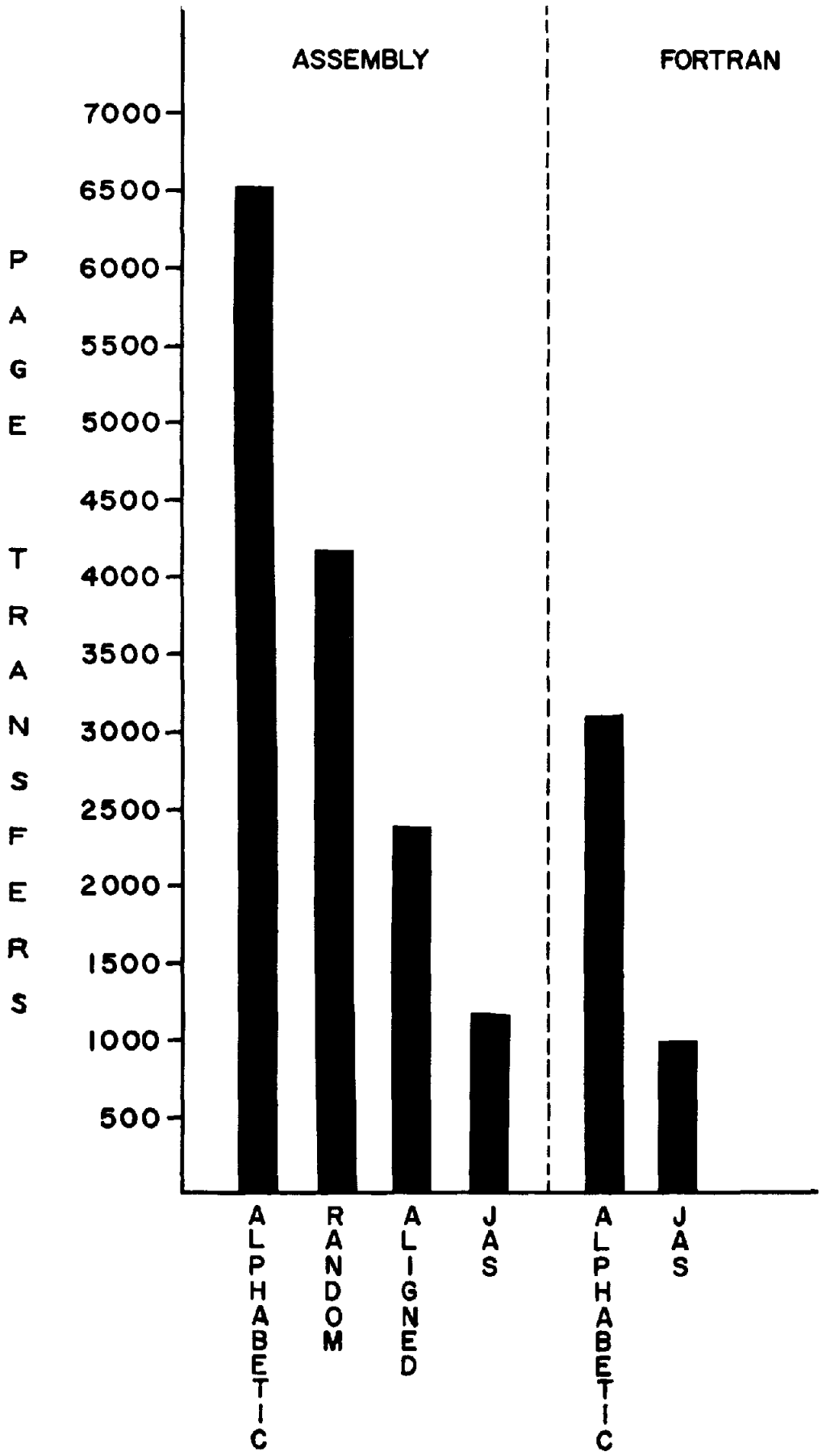
ASSEMBLY
FORTRAN
PAGE TRANSFERS
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
ALPHABETIC
RANDOM
ALIGNED
JAS
ALPHABETIC
JAS

FIGURE I.

APPENDIX A

PAGING STATISTICS PROGRAM

- PSTA -

This program takes the output tape created by CP/40[5] which is a record of significant information present when a demand for a new page is incurred in execution by the "virtual machine." The output is printed one line per page demand in the following manner:

INPAG	the page requested by the virtual machine
OUTPAG	the page picked by the algorithm to be overlaid by the incoming page
INSTR	the instruction counter of the "virtual machine" when the page demand occurs
Page Map	This is a record of the actual pages in the core storage when the demand occurred. The swap is already indicated. Page numbers proceed from left to right with each asterisk representing a page (Page numbers 0-3F). A plus indicates the used bit is one, a zero indicates used bit equals zero, and a space (blank) indicates that page is not now resident in core storage.

CORE RESIDENT PAGES (0-63) OF VIRTUAL MACHINE NUMBER 1

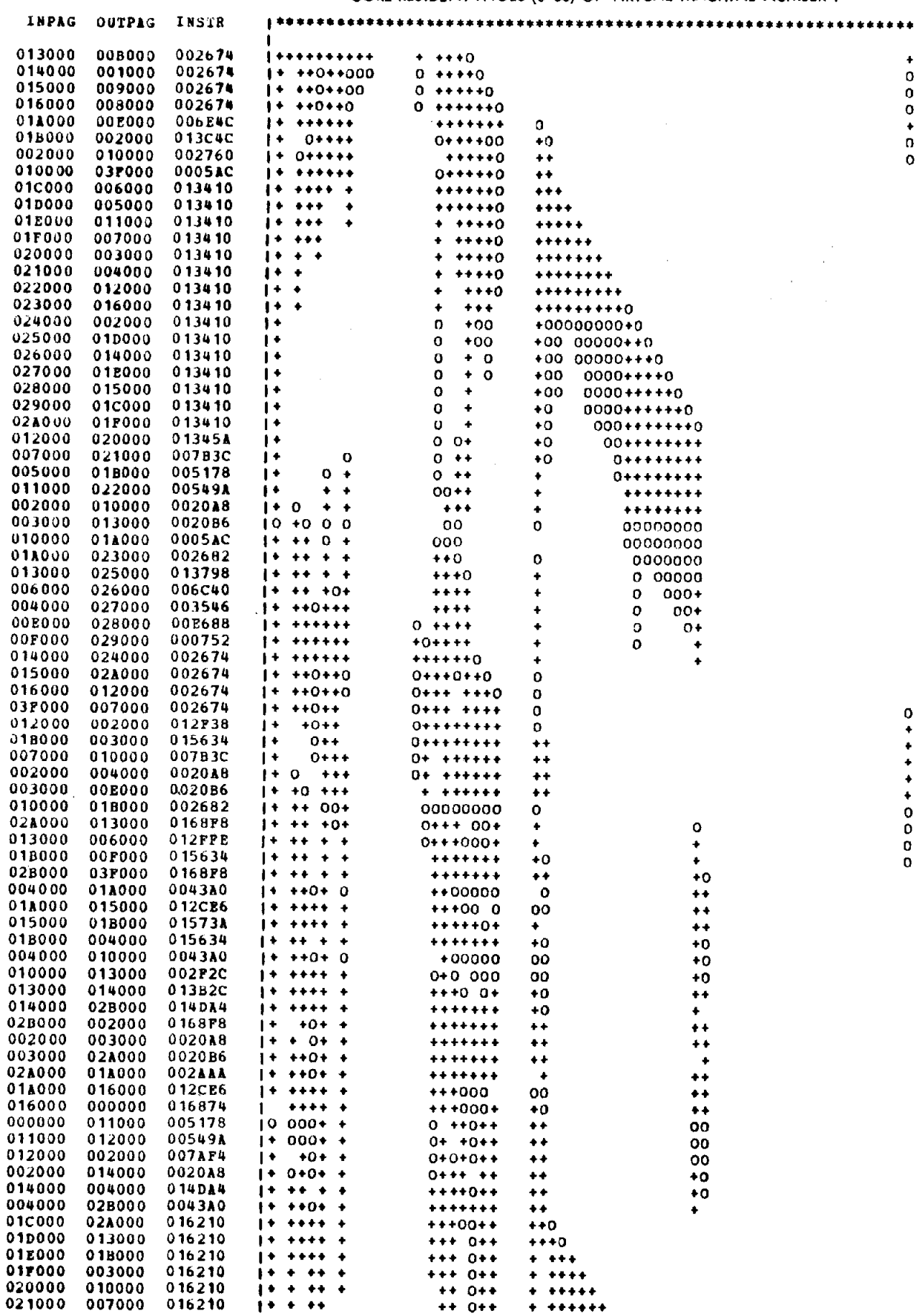

Fig. 2