Program behavior in a paging environment

by BARBARA S. BRAWN and FRANCES G. GUSTAVSON

IBM Watson Research Center
Yorktown Heights, New York

Study objectives

This paper is the result of a study conducted on the M44/44X system, an experimental time-shared paging system designed and implemented at IBM Research in Yorktown, New York. The system was in operation serving up to sixteen users simultaneously from early 1966 until May 1968. Conceived as a research project to implement the virtual machine concept, the system has provided a good deal of information relating to the feasibility of that concept.[1] The aim of this study is to investigate the concept more thoroughly from a user's viewpoint and to try to answer some important questions related to program behavior in a paging environment. As an experimental system, the M44/44X provided an excellent vehicle for the purposes of this study, and the study itself forms some basis for an evaluation of the system.

It is recognized by the authors that the results and conclusions presented here are to a great extent characterized by a particular configuration of a particular paging system, and as such do not constitute an exhaustive evaluation of paging systems or the virtual machine concept. Nonetheless, we feel that the implications of the conclusions reached here are of consequence to other system implementations involving paging.

Conventional vs automatic memory management

There has been much written about the benefits and/or disadvantages of paging machines and the virtual machine concept.[2,3,4] However, little data have been obtained which sheds a realistic light on the relative merits of such a system compared to a conventionally designed system. From a programming point of view there is little question that any technique which obviates the necessity for costly pre-planning of memory management is inherently desirable. The question that arises is—given such a technique, how efficiently is the automatic management carried out?

From a user's point of view this can simply mean—how long does it take to run a program which relies on the automatic memory management, and is this time comparable to the time it would take to run the program if it were written in a conventional way where the burden of memory management is the programmer's responsibility. It is this user's viewpoint that forms one focal point for this study.

The role of the programmer

Perhaps the most important aspect of the study concerns the role of the programmer. How does the role of the virtual machine programmer differ from that of the conventional programmer? For a conventional system the role of the programmer is well defined—the performance (i.e., running time) of his program is usually a direct result of his ability to make efficient use of system resources. How much he is willing to compromise efficiency for the sake of ease of programming may depend on how often the program is to be run. In any case, the decision rests with him. (There of course exist many applications where his choice of programming style or ability have little effect on performance; this case is of little interest to our study.)

When faced with the problem of insufficient machine resources to accommodate a direct solution of his problem, the conventional programmer is left with no choice but to use some procedure which is inherently a more complex programming task. The quality of the procedure he chooses may

have a dramatic effect on performance but it is at least a consistent effect and often quantifiable in advance. In any event, because conventional systems have been around a long time, there are many guidelines available to the programmer for achieving acceptable performance if he should wish to do so.

The role of the virtual machine programmer is not nearly so well defined. One of the original attributes claimed for the virtual machine concept was that it relieved the programmer from consideration of the environment on which his program was to be run. Thus he need not concern himself with machine limitations. As was pointed out previously, the question is—given that the programmer does in fact ignore all environmental considerations, what kind of efficiency results? Assuming that the answer to this question is sometimes undesirable, that is, running time is unacceptably long, another question arises. Can the programmer do anything about it? Clearly it is difficult to conceive of his being able to reorganize his program in such a way as to assure improved performance if he has no knowledge of the environment nor takes it into consideration when effecting such changes. Thus if the premise of freedom from environmental considerations is to be strictly adhered to, there can be no way for the programmer to consciously improve performance.

Should this premise be compromised to allow the programmer to influence performance through exercising knowledge of the system environment? This study assumes that this should indeed be the case and shows that there is much to be gained and little to be lost. It should be emphasized, however, that the original premise *need not* be compromised at all in so much as it would, of course, not be *necessary* for the programmer to ever assume the responsibility of having knowledge of the environment (unlike in the case of the conventional programmer faced with insufficient machine resources). It would only enable him to have better assurance of acceptable performance if he chose to do so.

Clearly, the many interesting questions concerning the role of the virtual machine programmer and his effect on performance are worthy of pursuit. We feel that the measurements obtained in this study of program behavior in a paged environment provide a valuable insight to such questions and serve as motivation for further consideration of them.

Test environment

Before discussing the results of the study we feel it is advisable to describe the environment in which they were obtained. Thus included herein are brief descriptions of the M44/44X system and the methods employed to obtain and measure the test load programs. (More complete information is available in References 1, 5 and 7.) It is assumed throughout this discussion that the reader is generally familiar with the concepts of virtual machines, paging, time-sharing and related topics; however, a short general discussion on paging characteristics of programs is included in order to establish an appropriate reference frame for the presentation of the experimental data.

The experimental M44/44X system

To the user a virtual computer appears to be a real computer having a precise, fixed description and an operating system which provides various user facilities and links him to the virtual machine in the same way as the operating system of a conventional system links him to the real machine (Figure 1). Supporting the virtual machine definition is a transformation (control) program

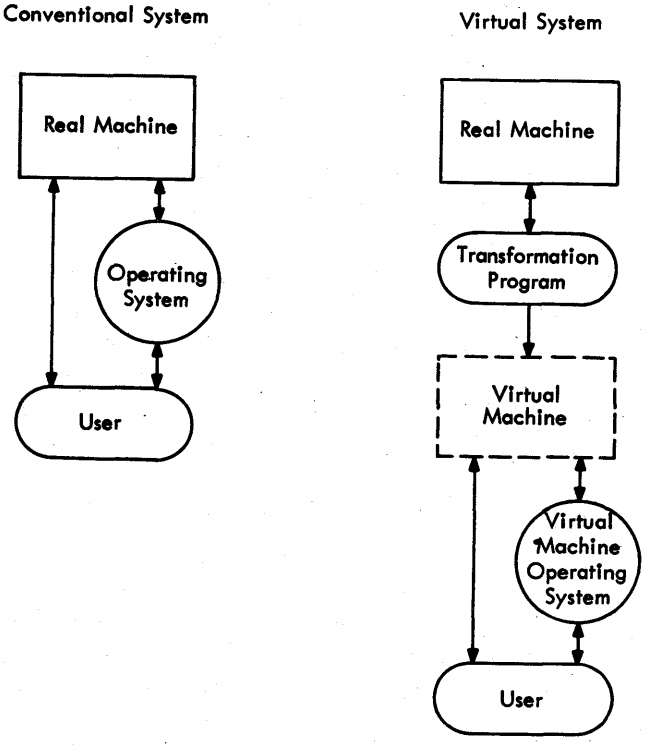

FIGURE 1—Conventional and virtual systems

which runs on the real machine. This program, together with special mapping hardware, "creates" the virtual machine as it appears to the user. Implementation of multi-programming within the framework of the virtual machine concept permits the transformation program to define the simultaneous existence of several separate and distinct virtual machines.

The virtual machine programmer may write programs without knowledge of the transformation program or the configuration of the real machine—his concern being the virtual machine description, which is unaffected by changes in the real hardware configuration or the transformation program. In the M44/44X system the real machine is called the M44, the transformation program is MOS, the virtual machine is the 44X and the virtual machine operating system is the 44X Operating System (Figure 2).

The real computer

Figure 3 shows the hardware configuration of the M44 computer. It is an IBM 7044 with 32K, 36-bit words of 2 μsec core which has been modified to accommodate an additional 184K words of 8 μsec core and a mapping device. The resident control program together with the mapping device and its associated 16K, 2 μsec mapping memory, implement the 44X virtual machines on a demand paging basis in the 8 μsec store. The back-up store of the M44/44X system, which is used for both paging and permanent file storage, consists of two IBM 1301 II disks. The page size (a variable parameter on this system) used for our tests was 1024 words (1 K). The average time required to seek and transmit one page from the disk to core is 0.21 second for that page size (computed from our data). The IBM 7750 serves as a message switching device, connecting a number of IBM 1050 terminals and teletype 33's to the system. To facilitate measurement our tests were not run from terminals (foreground) but as background jobs from tape. (The system makes no distinction between the two for the single programmed case—nor for the multi-programmed case as long as *all* jobs on the system are of the same type, i.e., all background or all foreground.)

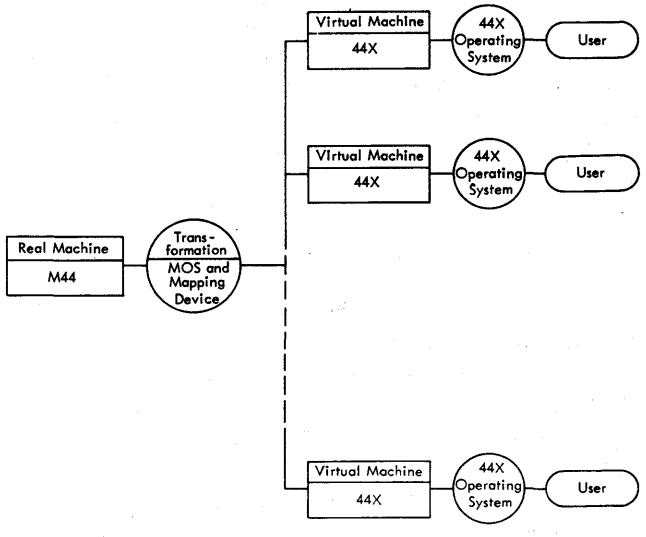

FIGURE 2—M44/44X Multi-programming system

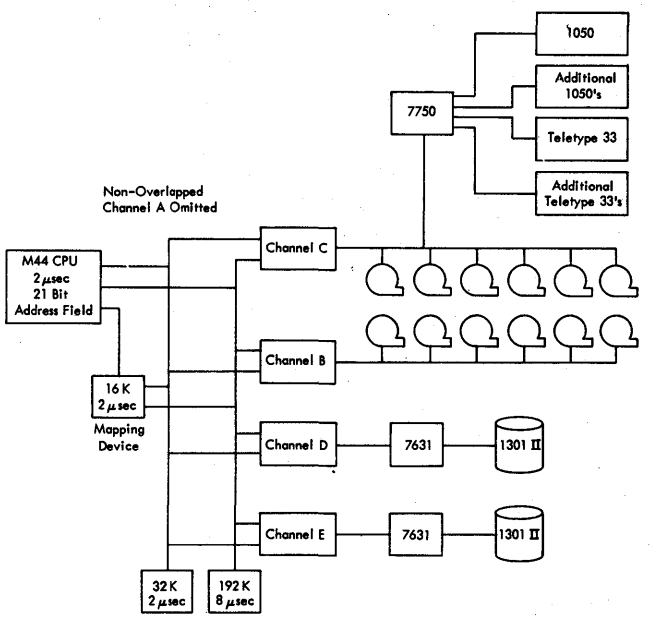

FIGURE 3—M44/44X hardware configuration

The control program

MOS, the control program, resides in the non-paged 2 μsec store. This M44 program "creates" and maintains each virtual 44X machine and enables several 44X's to run simultaneously, allocating the M44 resources among them. All 44X I/O is monitored by MOS, and all error checking and error recovery is performed by MOS. Some of the design parameters of MOS are easily changed to facilitate experimentation. The variable parameters include the page size, the size of execution store (real core) made available to the system, the page replacement algorithm, the time slice and the

scheduling discipline (via a load leveling facility). The last two parameters mentioned are applicable only in the multi-programmed case. As previously stated, the page size used throughout the study was 1024 words. The size of real core was, of course, one of the most important parameters and was varied to investigate paging properties of the programs (in both the single and multi-programmed environments).

For the single programmed part of the study the page replacement algorithm employed was FIFO (First In-First Out). If a page in real core must be overwritten, the page selected by FIFO is the one which has been in core for the longest period of time. Data were also obtained for single programmed paging behavior under a minimum page replacement algorithm developed by L. Belady.[6] A non-viable algorithm, MIN computes the minimum number of page pulls required by examining the entire sequence of program address references.

For the multi-programming part of the study, a time slice of 0.1 second was used. Runs were made using three different page replacement algorithms to determine the effect of this design parameter on system performance. (Available real store is competed for freely by all the 44X's.) The three algorithms were FIFO, BIFO, a biased version of FIFO which favors (on a round robin basis) one 44X by choosing not to overwrite the pages associated with it for a preselected interval of time, and AR, a hardware supported algorithm which chooses a candidate for replacement from the set of pages which have not been recently referenced. (These algorithms are described more fully in Refs. 1 and 6.)

The virtual machine

Each virtual 44X machine is defined to have 2^{21} words of addressable store. The virtual memory speed of a 44X is 10 μsec (44X programs are executed in 8 μsec store and a 2 μsec mapping cycle is added to a memory cycle); the CPU speed is 2 μsec. The user communicates with the 44X virtual machine through the 44X Operating System, a 44X program which permits continuous processing of a stack of 44X jobs; it contains a command language, debugging facilities, a FORTRAN IV compiler, an assembly program, a relocatable and absolute loader facility, routines for handling a user's permanent disk files and a subroutine library.

Test load problems

Test problems were chosen from the scientific, commercial, list processing and systems areas of computer applications. The problems chosen involved large data bases which required the programmer of a conventional machine to concern himself with memory management. The problems discussed in this paper include matrix inversion and data correlation from the scientific area and sorting from the commercial area. (A complete report on the entire study can be found in Ref. 7.)

Programs were initially coded for each problem in two ways:

i) a conventional manner where the burden of memory management is assumed by the programmer (conventional code), and
ii) a straightforward manner utilizing the large virtual memory ("casual" virtual code).

Simple modifications were then made to the "casual" virtual codes to produce programs better tailored to the paged environment. Our interest lay in comparing the performance of the different versions of the virtual codes under variable paging constraints in both single and multi-programming environments. We were also interested in comparing the conventionally coded program performance with that of the virtual (i.e., automatic memory management) codes given the same real memory constraints.

It should perhaps be noted here that for our purposes a program's performance is directly related to its elapsed run time. Thus in a paging environment, where this elapsed time includes the time necessary to accomplish the required paging activity, poor paging characteristics are reflected by increased run time and thus degraded performance.

Measurement techniques

A non-disruptive hardware monitoring device capable of measuring time spent in up to ten phases of program execution was used for all 7044 runs and relevant single-programmed 44X runs. In addition, for 44X runs (both single and multi-programmed), a software measurement routine in MOS was utilized. This routine collects data while the system is running (using the clock and a special high-speed hardware counter) and on system termination produces a summary of the

data including; total time, idle time, time spent in MOS (including idle time), number of page exceptions, page pulls, page pushes and other pertinent run data.

All programs were run in binary object form as background jobs residing on a system input tape; all output was written on tape. For the multi-programmed runs, a facility of MOS was used which permits several background jobs to be started simultaneously. For the single programmed study the 44X programs were first run and measured on the system with sufficient real core available to eliminate the need for paging; these same programs were then run (and measured) in a "squeezed core" environment, i.e., with insufficient real memory available, thus necessitating paging.

Program behavior under paging

Program performance on any paging system is directly related to its page demand characteristics. A program which behaves poorly accomplishes little on the CPU before making a reference to a page of its virtual address space that is not in real core and thus spends a good deal of time in page wait. A program which behaves well references storage in a more acceptable fashion, utilizing the CPU more effectively before referencing a page which must be brought in from back-up store. This characteristic of storage referencing is often referred to as a program's "locality of reference." [6] A program having "good" locality of reference is one whose storage reference pattern in time is more local than global in nature. For example, although a program in the course of its execution may reference a large number of different pages, if in any reasonable interval of (virtual) time, references are confined to only a small set of pages (not necessarily contiguous in the virtual address space), then it exhibits a desirable locality of reference. If, on the other hand, the size of the set is large, then the locality of reference is poor and paging behavior is correspondingly poor. (The "set" of pages referred to in the above example corresponds roughly to Denning's[8] notion of a "working set."

All programs typical of real problems exhibit badly deteriorated paging characteristics when run in some limited real space environment. What is of interest is the extent to which the space can be limited without *seriously* degrading performance. Clearly, the size of this space is related to the program's locality and provides some indication of the size of what might be called the program's critical or characteristic working set. As the single programmed results presented below show, the effects of programming style on the relative size of this space can be enormous.

Single programmed measurement results

We first measured the behavior of the 44X programs in a controlled single programmed environment. The results obtained are discussed in terms of the relative effects of programming style on performance for three problems: T1—Matrix Inversion, T2—Data Correlation, and T4—Sorting. In each case we are concerned with showing how even simple differences in programming technique can make a substantial difference in performance. Unquestionably there are further improvements which could be made in the algorithms employed; however, we feel that our point is best illustrated by the very simplicity of the changes made.

Timing and paging overhead data are given for actual runs made on the system employing a FIFO page replacement algorithm. Also, in order to establish that these results were not unduly influenced by that page replacement algorithm, corresponding computed minimum paging overhead data are given (obtained through interpretive program execution and application of L. Belady's[6] MIN algorithm).

The data collected for the comparison of the automatic and manual methods of memory management is also discussed in this section.

Problem T1 . . . Matrix inversion

The virtual machine codes for this program were written in FORTRAN IV and are intended to handle matrices of large order. They all employ an "in-core" technique since the large addressable virtual store permits the accommodation of large arrays (the burden of real memory management being assumed by the system through the automatic facility of paging). The curves in Figure 4 give the respective program run times as a function of real core size for the three different versions which were written for the virtual machine. These times are for inverting a matrix of order 100 (which is admittedly not an unusually large array, but sufficiently large to illustrate our point without requiring an impractical amount of CPU time).

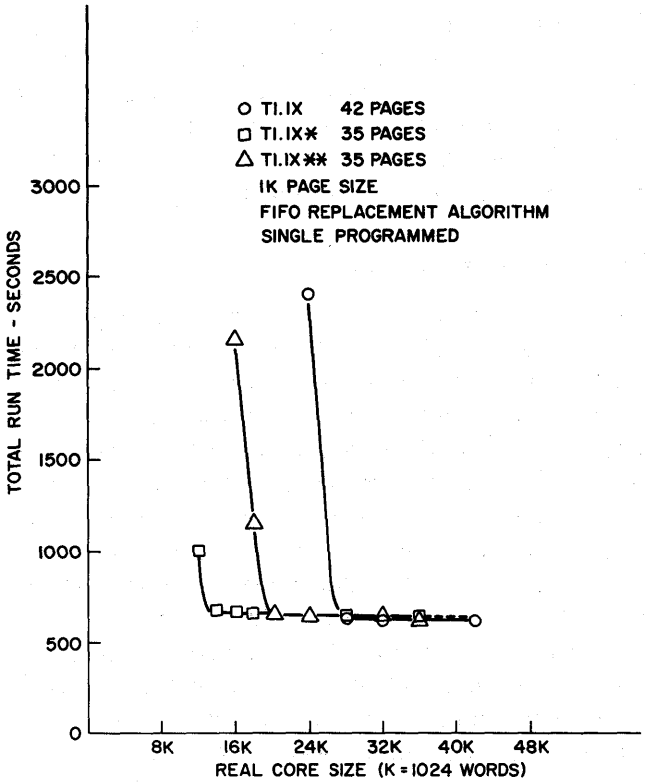

FIGURE 4—Effects of real core size
T1—Matrix inversion (100x100)

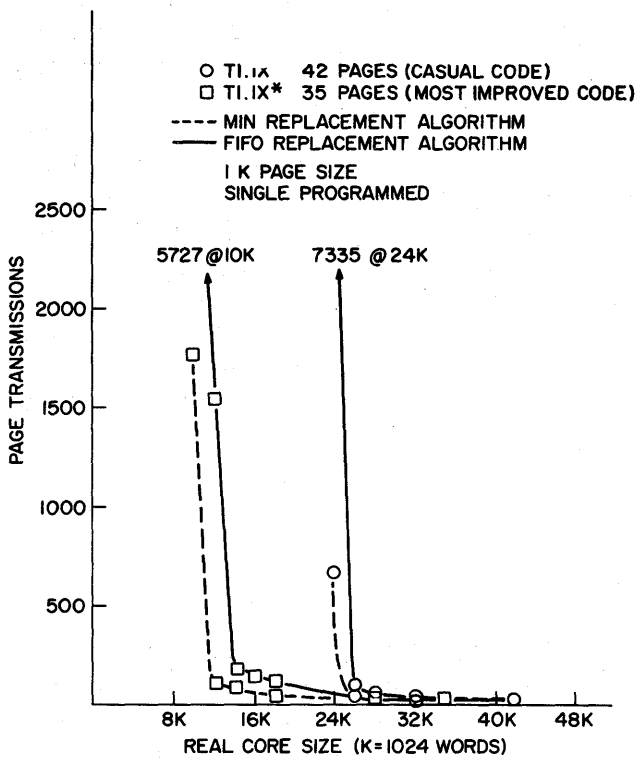

FIGURE 5—Effects of page replacement algorithm
T1—Matrix inversion (100x100)

All three programs employ the same algorithm, a Gaussian procedure utilizing a maximum pivotal condensation technique to order successive transformations. The differences in the three versions are extremely simple. The "casual" version, T1.1X, stores the matrix in a FORTRAN double subscripted array of fixed dimensions (storage allocated columnwise to accommodate a matrix of up to order 150), reads the input array by rows and prints out the inverted array by rows. The innermost computation loop traverses elements within a column. Version T1.1X** is the same as T1.1X except that variable dimension capability was employed (thus insuring the most compacted allocation of storage for any given input array). Version T1.1X* is the same as T1.1X** except that the input and ouput is columnwise instead of rowwise. Obviously neither of these changes is complicated or of any consequence in a conventional environment; however, as clearly shown in Figure 4, they make a considerable difference in a paging environment.

The paging overhead data is shown in Figure 5 for the casual (T1.1X) and the most improved (T1.1X*) versions for both the FIFO algorithm (corresponding to the time curves of Figure 4) and the MIN algorithm. This paging overhead is given in terms of the number of page transmissions required during execution of the respective program when run with a given amount of real core available under the discipline of the particular page replacement algorithm. (Each reference to a page not currently residing in real core requires a page to be transmitted from backup store into real core [a "pull"] and often also requires a page to be copied from real core onto backup store [a "push"]. The total number of pulls and pushes is the number of page transmissions. Given a particular real core size, the MIN algorithm employed gives the theoretical minimum number of pulls required. Belady has shown that the number of page transmissions obtained by this algorithm differs insignificantly from the number obtainable by minimizing both pulls and pushes.)

As can be seen in Figure 5, there is no great disparity between the paging overhead sustained under FIFO and the theoretical minimum possible (under MIN) for either of the programs. In particular it should be noted that the paging behavior of the well coded program is considerably better

under FIFO than that exhibited by the casual program under the most optimum of page replacement schemes. Certainly these data support the argument that improvement in programming style is advantageous to performance, irrespective of what page replacement scheme is used.

Clearly there are modifications which could be made to the algorithm itself which would further improve performance through improved locality of reference. McKellar and Coffman[9] have indeed shown that for very large arrays, storing (and subsequently referencing) the array in sub-matrix form (one sub-matrix to a page) is superior to the more conventional storage/reference procedure employed in our programs. (For the 100×100 array, however, the difference is not significant.)

Problem T2 . . . . Data correlation

For the other problem in the scientific area an existing conventional FORTRAN program, which required intermediate tape I/O facilities because of memory capacity limitations, was modified to be an "in-core" procedure for the virtual machine. The problem, essentially a data correlation procedure, involves reconstructing the most probable tracks of several ships participating in a joint exercise, given a large input data set consisting of reported relative and absolute position measurements. The solution implemented is a maximum likelihood technique; the likelihood functions relating the independent parameters are Taylor expanded to yield a set of simultaneous equations with approximate coefficients. The equations are solved (using the inversion procedure of problem T1), the solutions are used to recompute new approximate coefficients, and the process is reiterated until a convergent solution is reached. (Each iteration involves a single pass of the large data set.) The measured position data, together with the accepted solution are used to compute the reconstructed ships' tracks. (This final step requires one pass of the data set for each ship.)

For the first (or "casual") version, T2.1X, the conventional code was modified for the large virtual store in the most apparent way. The large data set, a mixture of fixed and floating point variables stored on tape for the conventional version, was stored in core in several single-subscripted fixed dimension arrays, one for each variable in the record format. As the curve for this program in Figure 6 shows, the performance is rather poor. This is accounted for in part by the fact that the

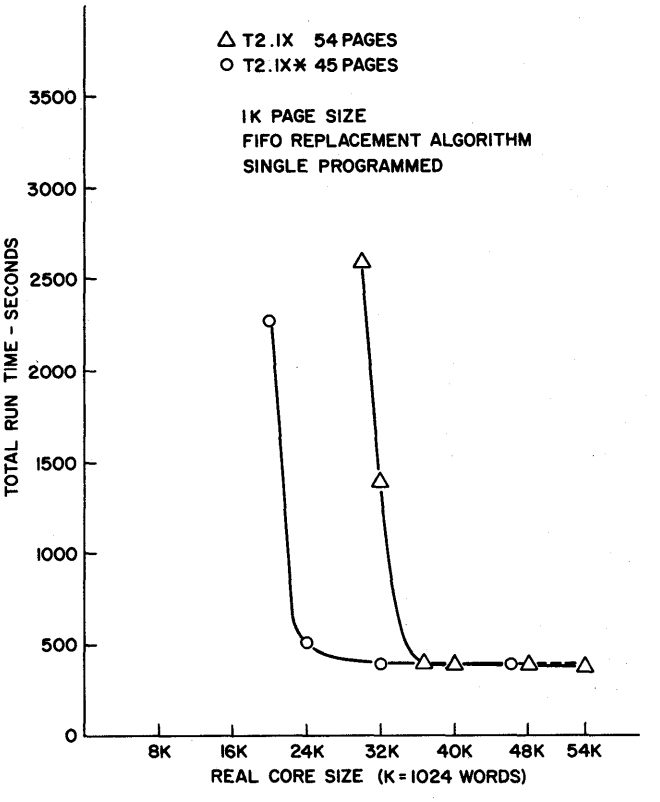

FIGURE 6—Effects of real core size
T2—Data correlation

manner in which the data are stored causes a global reference pattern to occur due to the program's logical use of those data. Version T2.IX* attempts to improve the locality by storing the data compactly in one single-subscripted floating point array, such that all of the parameters comprising a single logical tape record in the conventional code are in sequential locations. (The conversions necessitated by assigning both fixed and floating point variables to the same array name increased the CPU time slightly.) The curves in Figure 6 clearly show that this modification resulted in a significant improvement.

The same ordered relationship exhibited under FIFO holds for the casual and improved versions under the MIN algorithm (Figure 7). Although in the case of the poorly behaving code, the MIN algorithm does appreciably better than FIFO given a core size of 32K where FIFO performance has already deteriorated badly. The improvement is short lived, however, since deterioration under MIN occurs with any further decrease in real core size.

It should be noted that the actual data set used for these runs was not exceptionally large (as the total number of pages referenced indicates). Again, practicality demands that we settle for a data case of reasonable size. The case at hand involved six ships (resulting in 26 equations) and a rather small data base of only 240 reports. The data base storage requirements in the case of the well coded program, T2.1X*, were satisfied by four pages. In the case of T2.1X, however, the several large *fixed dimension arrays* used to store the data in that program required 13 pages; thus not only was the data ordering poor but a great deal of space was wasted as well.

Once again, there are probably other improvements that could be made. For example, because the program is divided into several subroutines (17) of reasonable length, a change in the order of loading the routines could improve (or degrade) performance. We have illustrated here only the effects of a change in the manner of storing the data base.

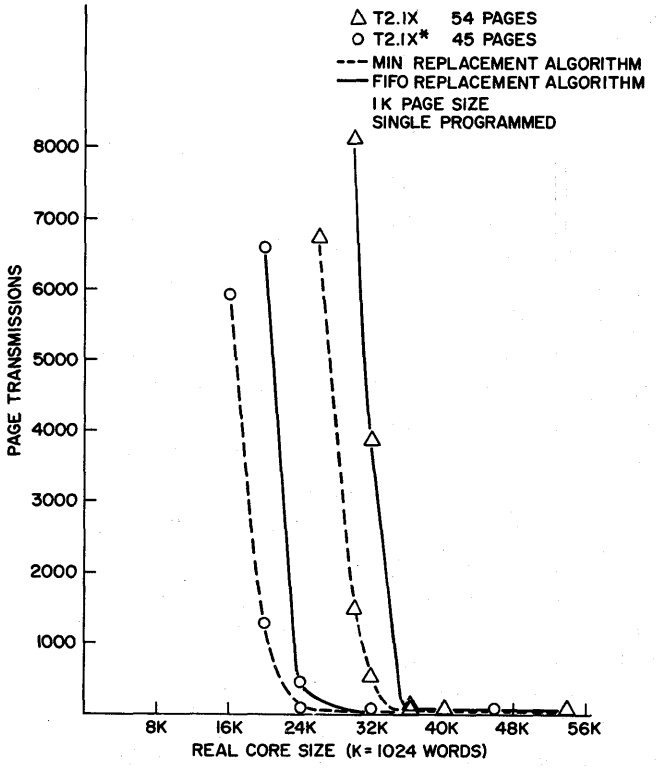

FIGURE 7—Effects of page replacement algorithm T2—Data correlation

Problem T4 . . . . Sorting

Sorting, a classical example of the necessity for introducing complicated programming techniques to accommodate a problem on a conventional memory bound computer, also affords an excellent example of how drastically programming style can effect performance in a paging environment. Ideally, if memory capacity were sufficient for the entire file to be in core, the sort programmer would only need to concern himself with the internal sorting algorithm and never be bothered with the other plaguing procedures involved with doing the job piecemeal. This was the approach taken, programming the virtual machine codes assuming that the file could be accommodated in virtual store.

Initially, two different algorithms were coded—the Binary Replacement algorithm (basically a binary search/insertion technique employed in a generalized sorting program in the Basic Programming Support for IBM System 360) and the Quicksort[10] algorithm (a partitioning exchange procedure). When the completed programs were run with a reasonably long data set, it became immediately apparent that the Binary Replacement algorithm was exceptionally bad for *large* lists because of the amount of CPU time required. (Note that this characteristic presents little problem for the internal sort phase of a conventional code which never deals with a very large list.) We will, of course, acknowledge that someone more knowledgeable in the field of sorting than we would have recognized this characteristic of the algorithm beforehand. Our experience nonetheless pointed out rather dramatically that an accepted technique for a conventional machine need not be acceptable when translated to a virtual machine environment, *irrespective of its paging behavior!* Because of its unacceptable CPU characteristics, the algorithm was discarded and our efforts were concentrated on Quicksort since that algorithm is efficient for either small or large lists.

Four versions were ultimately coded for the virtual machine, each of which is described below. All of the changes made to get from one version to another were simple and required little programmer time. None of these changes altered the total number of pages referenced; they simply improved the locality of reference. The time curves in Figure 8 and the paging curves in Figure 9

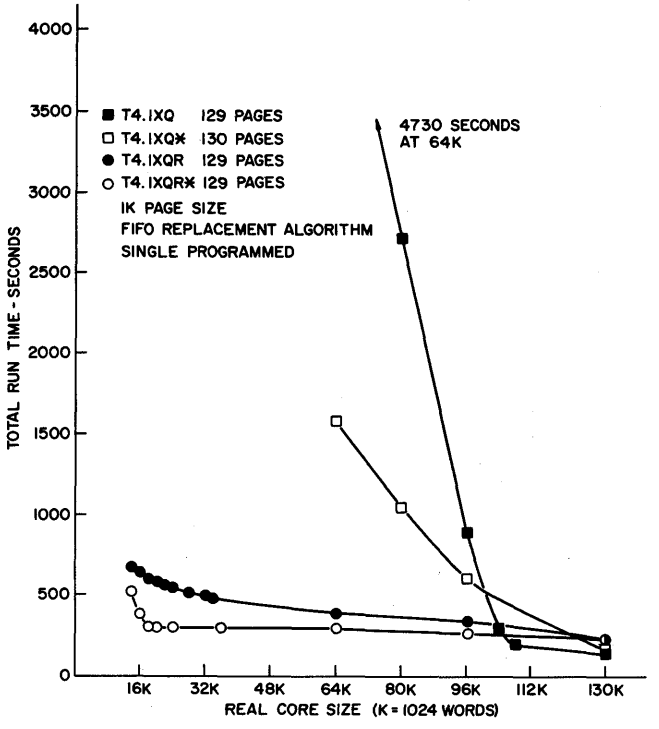

FIGURE 8—Effects of real core size
T4—Sorting (10,000 10-word items)

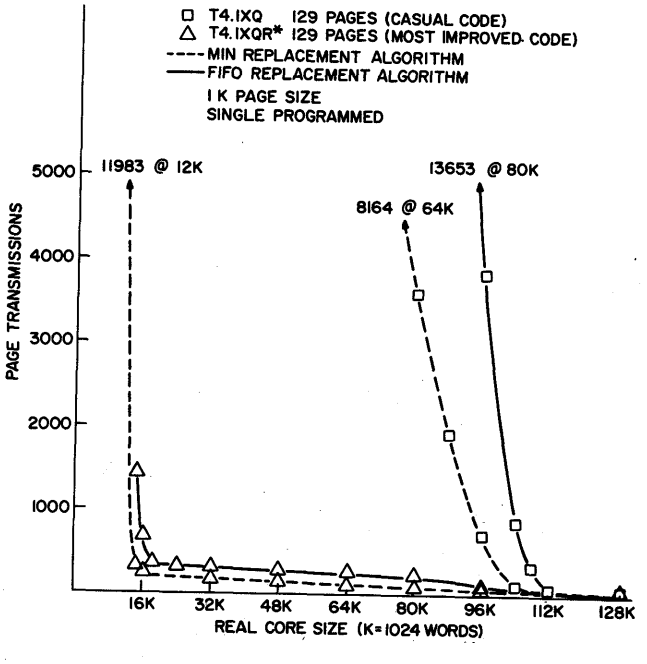

FIGURE 9—Effects of page replacement algorithm
T4—Sorting (10,000 10-word items)

show dramatically how important these relatively minor modifications were to performance. (The size of the file in the case shown is 100,000 words, occupying 100 pages in virtual memory.)

T4.1XQ, the "casually" coded version, reads in the entire file, performs a *non-detached keysort* utilizing the Quicksort algorithm and a table of key address pointers, then retrieves the records for output by using the rearranged table of pointers. The records themselves are *not* reordered during the sort thus storage references are random and global during both sort and retrieval, making locality of reference poor. Deprived of only a small amount of its required store, this program behaves very badly. Note that although the MIN curve in Figure 9 does show some improvement in paging behavior over FIFO, the improvement is of no consequence since performance is still quite unacceptable.

*T4.1XQ** treats the file as N sublists; each is read in, then sorted using the non-detached keysort routine of T4.1XQ. (N is 10 for the case shown; thus the 100 page file is logically divided into sublists of 10 pages each.) When all the sublists have been sorted an N-way internal merge, using the table of pointers, retrieves the records for output. This modification improves the locality of reference for the sort phase (for the case shown, the size of the characteristic working set during the sort is approximately 12; 10 for the current sublist, one instruction page and one page for pointers) but the storage reference pattern remains random during merge-retrieval since the records are not reordered within the ten sublists (during this phase the characteristic working set therefore includes *all* the file pages).

T4.1XQR is the same as T4.1XQ except that a record sort is performed instead of a keysort, i.e., *the records are reorderd* while being sorted, leaving an ordered list to be retrieved for output. The now sequential reference pattern substantially improves paging behavior for the retrieval phase (each page of the file is referenced only once for that phase). Moreover, because of the record reordering, locality during the sort phase benefits substantially from the partitioning characteristic of Quicksort. Performing a record sort, of course, results in a penalty in CPU time (especially for large records) since the transfers involve the entire record instead of the key only. (For this reason, it would not be wise to choose a sorting algorithm which requires an exceedingly large number of transfers.) However, the penalty paid is

relatively insignificant in view of the improved paging behavior.

*T4.1XQR** is the same as T4.1XQ* except that it performs a record sort instead of a keysort. The comments made on improved locality during the sort phase for both T4.1XQ* and T4.1XQR also apply to this version. In addition, because the records are now in sequential order within each sort area (due to record reordering) the merge/retrieval phase also exhibits desirable paging characteristics (as long as there are enough pages available to accommodate *at least* one page from each sort area, i.e. N pages plus instruction and control pages—approximately 13 for the case shown).

Clearly, the behavior of T4.1XQR and T4.IXQR* demonstrates that the large virtual store can be used effectively and in a simple manner if thought is given to the environment. The curves for T4.1XQ and T4.1XQ* demonstrate equally clearly that it can be disastrous not to do so.

Automatic vs programmer-controlled memory management

The objective of this part of the study was to compare the effectiveness of automatic and manual (programmer-controlled) memory management. To meet this objective, our test problems were programmed to run on a conventional machine, using accepted manual methods to accommodate them on the available memory. The efficiency of any program written for a conventional machine, of course, depends on how skillful the programmer is in utilizing available system resources. We felt that, although in no way optimum, the efficiency of the programs coded was characteristic of what is normally achieved under practical constraints of programmer time. (It should be noted that the programmer time involved in writing and debugging these conventional codes far exceeded that required for the corresponding virtual codes.)

The data presented in the previous sections clearly show that the effectiveness of paging as an automatic memory management facility depends not only on internal characteristics of the particular system but also on user programming style. We thus felt that an effective comparison of the two memory management methods should include the effects of virtual machine programming style. We also felt that our comparison should in some way include the effect which overlap capability can have on conventional code efficiency since, in a multi-programming environment, that capability does not exist for the individual user. (We were aware that almost any proposed comparison would be subject to question on one count or another because of the lack of adequate control; we nonetheless feel that the comparison is reasonably unbaised and has sufficient validity to be of interest.)

To make the comparison we proposed to run both the conventional programs and the corresponding virtual programs (i.e., those which utilized the large virtual store), in their respective environments, which were constrained to be equivalent with respect to real machine resources. All of the conventional program I/O was tape I/O and the CPU and memory speed were the same for both the conventional and virtual machines. In each case the virtual programs were run with the size of available real core equal (to the nearest page) to that actually referenced by the corresponding conventional program. The numbers in Table 1 are computed ratios of the respective virtual code run times to corresponding conventional code run times; therefore numbers less than 1 are favorable to the automatic method.

TABLE 1—Comparison of automatic and programmer controlled memory management

	Casual Virtual Code vs Overlapped Conventional Code	Best Virtual Code vs Overlapped Conventional Code	Casual Virtual Code vs Non-Overlapped Conventional Code	Best Virtual Code vs Non-Overlapped Conventional Code
T1 - Matrix Inversion 200 x 200	2.33	1.25	1.47	0.79
T2 - Data Correlation	0.81	0.81	0.71	0.71
T4 - Quicksort 100,000 word file [4-way Merge] [Conv. Code]	>41	1.26	>32	0.99
T4 - Quicksort 100,000 word file [2-way Merge] [Conv. Code]	>27	0.84	>19	0.59
T4 - Quicksort 1,000,000 word file [4-way Merge] [Conv. Code]				1.15

The data indicate that, if reasonable programming techniques are employed, the automatic paging facility compares reasonably well (even favorably in some instances) with programmer controlled methods. While not spectacular, these results nonetheless look good in view of the substantial savings in programmer time and de-

bugging time that can still be realized even when constrained to employing reasonable virtual machine programming methods.

Multi-programming measurements

The importance of programming style to paging behavior was clearly demonstrated in the single programmed part of this study. We were interested in learning if it would have similarly dramatic effects on performance in the domain more common to paging systems, i.e., multi-programming. Because the most notable changes in behavior were observed in the sorting area, we decided to plan our measurement efforts around these programs. An extensive measurement program was undertaken which was designed to give us insight into the relative effects on performance of the following: programming style, page replacement algorithm, size of real core, number of users and scheduling. It should be noted that the question of performance in a multi-programmed environment involves both the individual user response and total system thruput capability. Although the study addressed both of these aspects, the results discussed here pertain only to the latter. (A complete in-depth report on the entire multi-programmed measurement study is given in Ref. 7, Part III.)

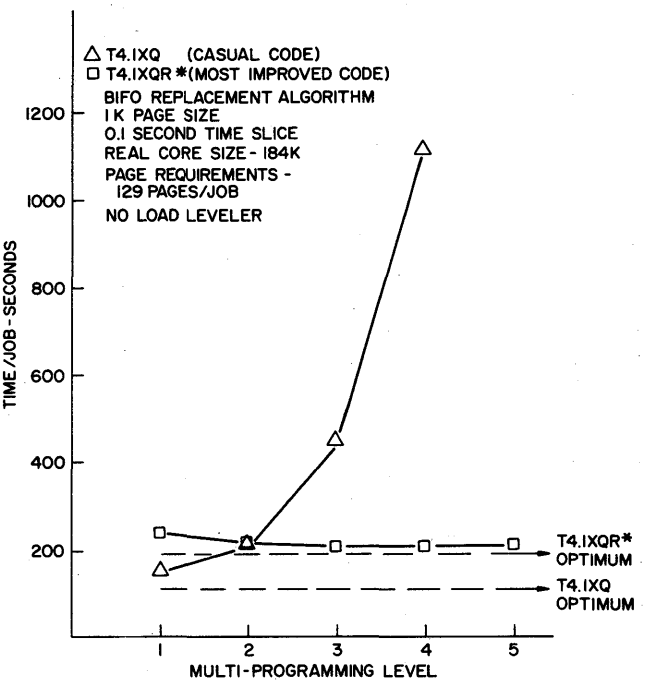

FIGURE 10—Effects of programming style
T4—Sorting (10,000 10-word items)

The effects of programming style

The two versions of the sort program used for this study were the "casually coded" version, T4.1XQ, and the "most improved" version, T4.1XQR*. Multiple copies of a given program were run simultaneously (as background jobs) on the system with the full real core (184K) available. (No more than 5 background jobs can be run simultaneously because of tape drive limitations.) The curves in Figure 10 compare the multi-programming efficiency obtained with the two different programming styles. These curves are plots of Time/Job vs the number of (identical) jobs run simultaneously on the system (Multi-programming Level).

Clearly the efficiency of the system is nearly identical whether multi-programmed at the two level or the five-level in the case of the well-coded program, T4.1XQR*, but is substantially degraded for each additional job in the case of the casually coded version, T4.1XQ. In fact, multi-programming at even the two level for that program is worse than running sequentially. (For T4.1XQR* multi-programming is consistently more advantageous than running sequentially up through the five-level.)

The effects of load leveling

One of the capabilities available on the M44/44X system aimed at improving efficiency is that of dynamically adjusting the load on the system in order to attempt to avoid the overload condition which is characterized by excessive paging coupled with low CPU utilization. When this load leveling function is activated, the system periodically samples paging rate and CPU utilization, compares them with pre-set criteria to determine if a condition of overload or underload exists, and then takes action appropriately to adjust the system load by either setting aside a user, i.e., removing him temporarily from the CPU queue, or restoring to the queue a user who was previously set aside. The function of the load leveler is thus essentially one which affects scheduling.

The extremely poor behavior exhibited by the casual code when multi-programmed made this case a likely candidate for studying the effects of load leveling. Figure 11 shows the remarkable improvement which the load leveler achieved

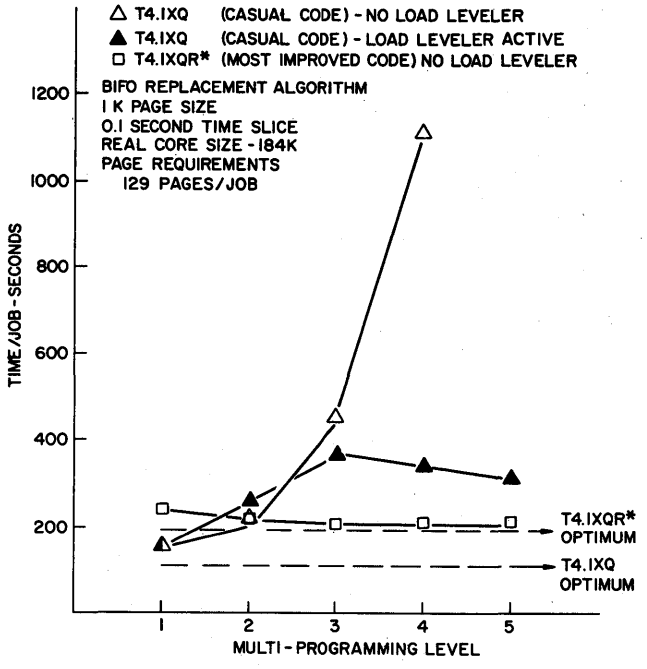

FIGURE 11—Effects of load leveling
T4—Sorting (10,000 10-word items)

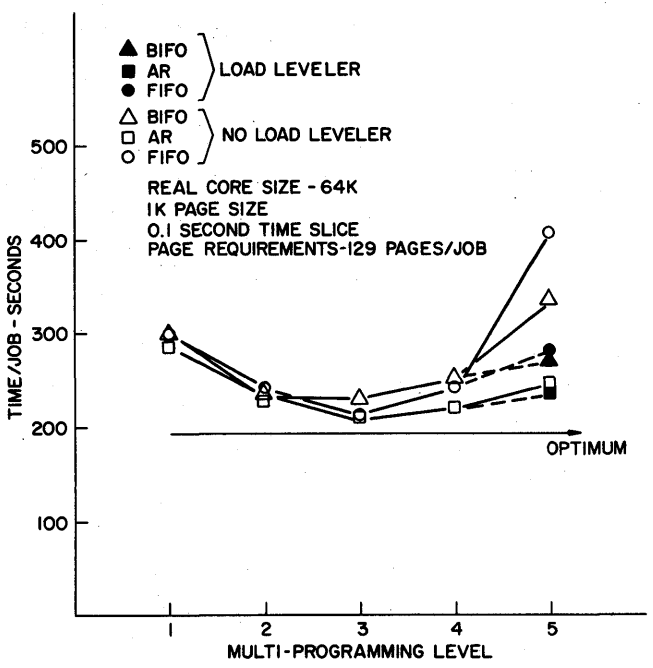

FIGURE 12—Effects of page replacement algorithm
T4.1XQR*

when there were three or more jobs involved. Unfortunately, the efficiency is still substantially worse than in the sequential case. We nonetheless feel that the potential for improved performance achieved through the use of an automatic dynamic facility such as this is promising and indicative that it would be well worth implementing—in particular if it can be kept simple and efficient as is the case with the M44/44X load leveler.

The effects of page replacement algorithm

As might have been suspected from the single-programmed MIN study, the role of the page replacement algorithm appears to be of relatively little significance. In the case of T4.1XQ, runs were made using the more sophisticated AR algorithm but the data collected differed little from that obtained for the BIFO algorithm. Similarly, in the case of T4.1XQR* the difference in the results is inconsequential for those runs made where all of real core (184K) was available. (Figures 10 and 11 show the BIFO data.) However, when the same T4.1XQR* runs were made with the real core size restricted to 64K there was some change in performance for the different replacement algorithms. The curves in Figure 12 compare the effects of using the different algorithms for T4.1XQR* multi-programmed (up to the five-level) with only 64K of real memory available to the entire system, i.e., shared by all the users. The level of multi-programming for which the efficiency is optimum is in all cases three; however, in the case of the AR algorithm, multi-programming at the five-level with only 64K of real memory is still more advantageous than running the five jobs sequentially (with the same 64K of real memory). Note that this is also true when running under the other algorithms with load leveling.

The effects of real size

Performance is so poor for the T4.1XQ program given the full 184K of real memory, that it is obviously unnecesary to show how bad things would be given an even smaller memory! In the case of T4.1XQR* however, performance for the system is so close to optimum that we were curious to learn just how small the real core size could be before performance would be worse than in the single-programmed case (for the same size of real memory). The curves in Figure 13 compare Time/Job for the single-programmed case, with multi-programming at the three and five levels for different real core sizes. Runs were also made multi-programmed at the five level with the load

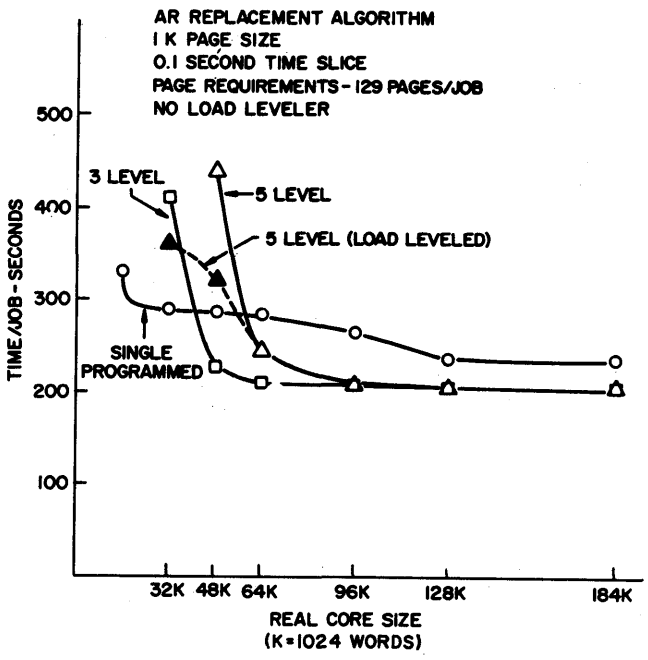

FIGURE 13—Effects of real core size
T4.1XQR*

leveler activated and real core sizes of 48K and 32K. As can be seen in Figure 13, while improving performance, the load leveler was not able to improve it sufficiently to compare favorably with the single-programmed sequential case.

When viewed in the perspective of page requirements per job, the performance of the system is remarkable for the well coded program. Five jobs, each requiring 129 pages, shared a 32K memory and still behaved reasonably well! (The time per job is even a few seconds less than that required for the overlapped 2-way merge conventional code.) On the other hand, the performance for the casual code given the full memory capability of 184K is at best (load leveled) quite a lot worse than sequential and at worst (not load leveled) a minor disaster.

The data which we have presented here on multi-programming represent only part of that collected for the study. The cases chosen are obviously the extreme ends of the spectrum. One would not (hopefully) encounter all "bad" programs running at the same time on a system under real time-sharing conditions, nor (regretfully) is one likely to encounter all "good" programs. The real situation lies somewhere inbetween—and, most likely, so does the characteristic performance of the system. We have not directly addressed the question of individual thruput (or response) time in the data shown here; however; we have shown that total system thruput is most certainly affected by the programming style employed by the users on the system. We have shown in our other work (Ref. 7) that this is also true for individual response time (often even if system thruput is unaffected).

SUMMARY

The single programmed data presented in this paper give strong support to the conclusion that the effects of programming style are of significant consequence to the question of good performance in a paging system. Indeed, as the MIN results indicate, the basically external consideration of programming style can be considerably more important than the internal systems design consideration of replacement algorithm. We feel that data obtained for the multi-programmed case, some of which were presented in the previous section, further support our conclusions. In view of these results, we feel that this aspect of performance must not be disregarded in future endeavors to implement paging systems. Programming techniques should be developed at both the user and system levels which are aimed at achieving acceptable performance on such systems. For example, higher level language processors such as FORTRAN should be designed for paging systems to produce good code for the environment as well as to perform well themselves in that environment.

While we support the stand that paging and virtual machines are inherently desirable concepts with much potential, we strongly feel that in order to fully realize that potential in terms of practical performance characteristics, the notion of programming with complete unconcern for the environment must be discarded. Our data have shown, however, that one can often realize acceptable performance by employing even simple techniques which acknowledge the paging environment. Their simplicity leads us to feel that the programming advantages inherent to the concept of virtual systems can, to a great extent, still be preserved.

ACKNOWLEDGMENT

We would like to acknowledge E. S. Mankin for his extensive contribution in preparing the test load programs for the sort area.

REFERENCES

1 R W O'NEILL
Experience using a time-shared multiprogramming system with dynamic address relocation hardware
SJCC Proceedings Vol 30 1967 pp 611–621

2 P WEGNER
Machine organization for multiprogramming
Proceedings of 22nd ACM National Conference Washington DC 1967 ACM Publication P-67 pp 135-150

3 G H FINE C W JACKSON P V MCISAAC
Dynamic program behavior under paging
Proceedings of 21st ACM National Conference Washington DC 1966 ACM Publication P–66 pp 223–228

4 *Adding computers—Virtually*
Computing Report for the Scientist and Engineer Vol III No 2 March 1967 pp 12–15

5 *The M44/44X user's guide* and the *44X reference manual*
IBM Corp T J Watson Research Center Yorktown Heights New York September 1967

6 L A BELADY
A study of replacement algorithms for a virtual storage computer
IBM System's Journal Vol 5 2 1966 pp 78–101

7 B S BRAWN F G GUSTAVSON
An evaluation of program performance on the M44/44X system Parts I II III
R C 2083 IBM T J Watson Research Center Yorktown Heights May 1968

8 P J DENNING
Working set model for program behavior
CACM Vol 11 5 May 1968 pp 323–333

9 A C McKELLAR E G COFFMAN
The organization of matrices and matrix operations in a paged multiprogramming environment
Princeton University Technical Report No 59 February 1968

10 C A R HOARE
Quicksort
Computer Journal Vol 5 April 1962 to January 1963 pp 10–15