An analysis of paging and program behaviour

M. Joseph*

* Computer Group, Tata Institute of Fundamental Research, Colaba, Bombay 5

The paper describes some analyses and simulations of the operation of programs in a paged memory. Patterns of page usage that are observed indicate that smaller page sizes may lead to a more economical implementation. Some predictive algorithms for paging-in are evaluated for their ability to reduce page faults.

(Received January 1969)

1. Introduction

Paging has become one established method of administering the store in either a multiprogramming environment or where the address space available to a program is greater than that present in the main memory (this was the case on ATLAS 1 and the term 'virtual memory' has been used by Belady (1966) to emphasise that the total address space need never all be present in the main memory). Such a scheme makes the assumption that few programs require access to their entire address space within small intervals of time so that it is possible to keep only the relevant portions, or pages, in the main memory; in fact, where it was felt that this criterion did not hold (Fine, McIsaac and Jackson (1966)) the usefulness of paging has been questioned strongly.

At the same time, the effect of paging the storage address space has been, to quote Varian and Coffman (1967), to change 'the logical units of information transfer' from 'the variable length program and data structures to the fixed length pages into which programs and data are fitted'. This is a slightly different position to that adopted in the original ATLAS 1 paging scheme (see Kilburn, Edwards, Lanigan and Sumner (1962)) as it has recognised that the effect of paging (in terms of the speed of execution of a program) is discernible to the programmer and that efforts have to be made to organise programs into forms which fit into fixed length pages. In this connection it is interesting to note that in MULTICS the concept of segmentation has been used to force the programmer to be aware of page boundaries by making the segment the normal unit of data storage: as segments contain only whole pages, this prevents the starting of a storage block at points within the page boundaries.

Another change since the ATLAS 1 implementation has been that, in general, paging is used to create virtual memory environments in multiprogrammed machines where the dynamics of store allocation are produced by a variety of independent processes (including the system supervisor). The original approach of letting the program grow to use up to a fixed amount of storage and then overlaying is altered and there are a variety of possibilities regarding replacement of pages which depend partly on the storage allocation procedure adopted: if each program in the main store is restricted to using a fixed amount of store, the process of replacement is normally activated by new page demands by that particular program; if the store restrictions are not as rigid, replacement of a page belonging to one program may be forced by any program making a new page demand. Without further refinement, both methods have disadvantages: in the first case, a program which tended to use, at a time, one or more pages above its fixed allocation would run inefficiently under most replacement algorithms and, in the second case, one could visualise a situation where, say, two programs in the main store alternately replaced pages belonging to each other, and which were required, and so almost indefinitely held up the machine. There have been suggestions, therefore, that programs be allowed to specify which pages they have dispensed with (or, alternatively, which pages they will need).

2. Areas of activity

Consider the total address space available to a program: the requests for access to this space come for instructions and operands, and without prior knowledge (and excluding those cases where by convention the first instruction to be executed is in the first location) the choice of the first instruction address is completely unpredictable and possibly even pseudo-random. Following this choice, however, the chances are considerably higher that the next instruction address will bear a simple relation to the first and, very often, that it will be adjacent to the first. In very simple cases the sequence of instruction addresses will be random-sequential, i.e. that the choice of the first address will be random and that a few subsequent addresses will then follow sequentially; but in practice, the sequence of instruction addresses is usually the ensemble of some repeated strings of sequential or near-sequential orders held together by small non-iterative connecting paths and, as far as the memory system is concerned, these addresses are interspersed with operand addresses. The relative predictability of operand addresses is often dependent on the complexity of the address arithmetic that a program performs in selecting its operands, but these too may be sequential to some degree (an obvious exception would be operand addresses chosen by a hash-coding procedure). Further, operand calls, or the processes of data analysis which make these calls, often consist of comparisons and/or arithmetic operations by one operand on another so that there are two or more strings of independently (semi) coherent addresses which combine to form the sequence

Downloaded from https://academic.oup.com/comjnl/article/13/1/48/388992 by guest on 19 April 2024

of operand addresses (an example of this would be a matrix multiplication routine which generated calls alternately to elements of the different matrices so that while calls to the storage area of any one matrix would appear coherent, the interspersed addresses in the order in which they appeared would not). Finally, there is the output catchment area where results are held prior to their being output. We have thus about four areas of activity within the address space: this corresponds to a suggestion by Gibson (1966).

In order to test the validity of this empirical argument and to provide quantitative estimates of the sizes of these areas of activity a series of simulations was made on address sequences collected from a number of programs performing various operations (see Appendix).

The simulation program read the address at the top of the sequence (the 'current' address), masked it down by the size of the page and then searched through a stack of the last used pages to find when the page was previously in use. If a match was found, a counter corresponding to that particular position in the stack was incremented, the current page (i.e. the page containing the current address) was put on the top of the stack and the other pages were pushed down one position. In the searching, it was also tested if the required page was adjacent to any others in the stack: if the condition was fulfilled, a counter for that position in the adjacency stack was incremented.

Fig. 1 shows the variation in the percentage of accesses to the *n*th last used page as *n* goes from 1 (i.e. the current page) to 7 (i.e. that page following the last use of which 6 other pages have been used). For two different sizes of page (32w and 1024w) the two curves appear remarkably close and it is seen that they both come down fairly sharply between $n = 1$ and $n = 4$. At $n = 4$ they level off, indicating that the earlier suggestion that there are four localities of activity within a program's address space was roughly correct, with the additional comment that these areas are fairly small.

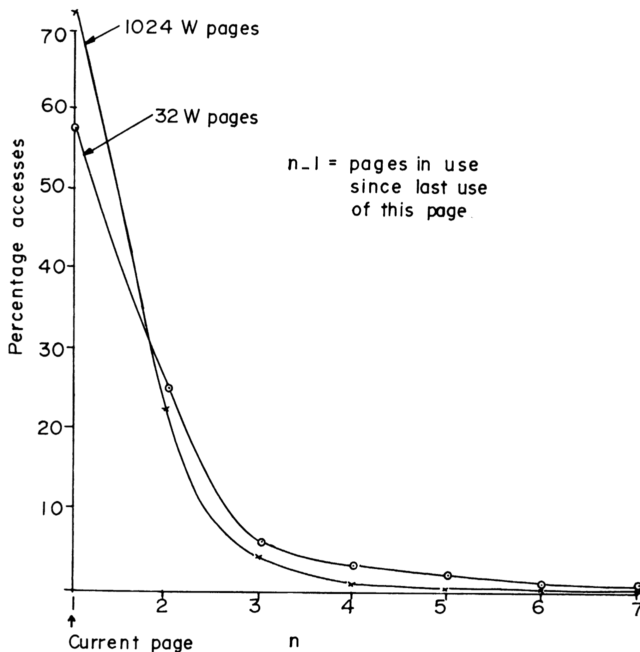

Fig. 1. Variation in percentage of accesses to the *n*th last used page

In **Fig. 2** we see the percentage of failed store accesses with different numbers of pages in store and for three different page sizes (32w, 128w and 1024w): in this figure the failure rate is plotted on a logarithmic scale. The separate lines, dotted and solid, in each case are for short runs (18 programs) and for long runs (2 programs) and it is seen that there is not a substantial difference. What is notable is that, for example, 5 pages of 32w bring the failure rate to a lower level than 2 pages of 1024w and that it is really only after the number of 1024w pages exceeds 5 (i.e. 5k of an average program size of 8k) that they show a marked improvement over the smaller pages.

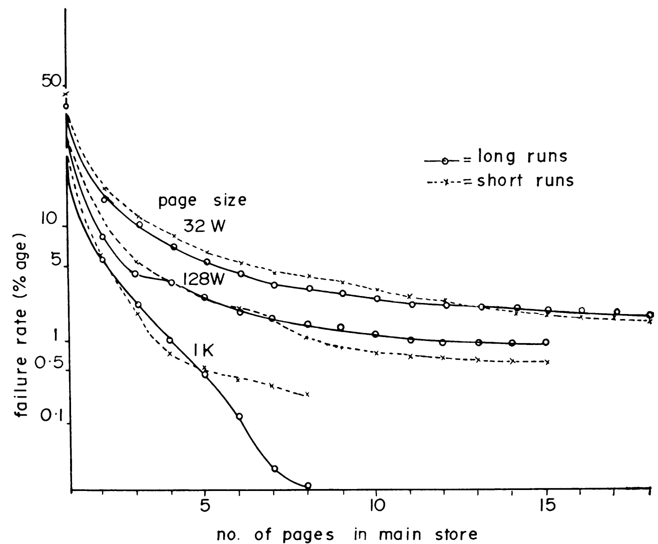

Fig. 2. Percentage of failed store accesses

If we plot the variation in failure rate with page size for a fixed number of words in store (see **Fig. 3**) it shows clearly that while for small allocations of store it is probably more efficient to use 32w pages, above 4k the larger pages tend to become more efficient.

Belady (1966), Varian and Coffman (1967), Scherr (1966) and others have pointed out some of the advantages of using small pages and these results would corroborate their evidence as it is seen that unless a substantial portion (i.e. half or more) of the program's address space is loaded, the same failure rate as for 1k pages can be obtained with the use of considerably less store by smaller pages. In the worst case, 16 times as many transfers would be required to load the same amount of information with 64w pages as with 1024w pages (i.e., there would be 16 times as many page faults just for loading): if, in spite of this, the failure rate is equal for 64w and 1024w pages it would indicate the subsequent number of page faults is correspondingly higher for 1024w pages.

3. Backing store delays

It has recently become clear (Nielsen (1967), Lauer (1967)) that the primary limiting factor in a multiprog-

Downloaded from https://academic.oup.com/comjnl/article/13/1/48/388992 by guest on 19 April 2024

rammed paging scheme is the possible rate of transfer of information from the backing store and the large access time (where discs or drums are used). It is often possible, in these cases, to partly overlap the access time for one transfer by other transfers, though this will mean extra computation time for the supervisor and may also mean inefficient use of the backing store, since efforts will have to be taken to ensure the relative dispersal of information on the backing store. If A is the access time of the backing store, P is the page size and N the number of words that can be transferred per unit time from the backing store, then the average delay for a new page demand will be $2(A + P/N)$, since for every page brought into the main store it is likely that a page will have to be sent to the backing store. If A is small (i.e. for a mass core store), the main delay is for the transfer time, $2P/N$, which is smaller for small pages. Another effect of backing store delay is that the area of core of a program waiting for a transfer is effectively lost to the system until the transfer is complete so that, with several programs held up, it is possible that the system will have to attempt to work without a large proportion of its main store. Hence, for this reason also, it is desirable that as few programs are held up as possible: if the access time to the backing store is large, there is little doubt that large pages will be necessary but, in other cases, it could be argued that the store saved by using small pages would be sufficient to allow the number of programs which are held up to increase without ill effect.

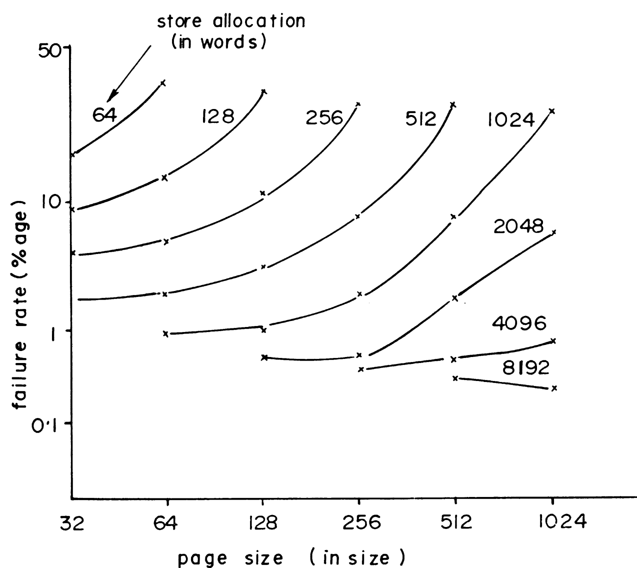

Fig. 3. Percentage of failed store accesses with different store allocations

4. Use of address space: two phase operation

Fine *et al.* (1966) have drawn up a graph of the increase in the use of store with time by a program and this, together with some results given below, would tend to indicate that most programs operate in two distinct phases: the first, in which they have a very large rate of increase in the use of store and, the second, where they make relatively few demands for new pages. If such a distinction can be made generally it would be undesirable for a system to have several programs starting within a small interval of time as a high increase in address space use (with the corresponding delays at every page fault and the increasing length of the backing store request queue) can only be tolerated when superimposed on a background of jobs which are in their second phase. And several studies of the use of replacement algorithms (e.g. Varian and Coffman (1967)) have shown that in the first phase the program requires a large number of new pages with this requirement being generally insensitive to the page size.

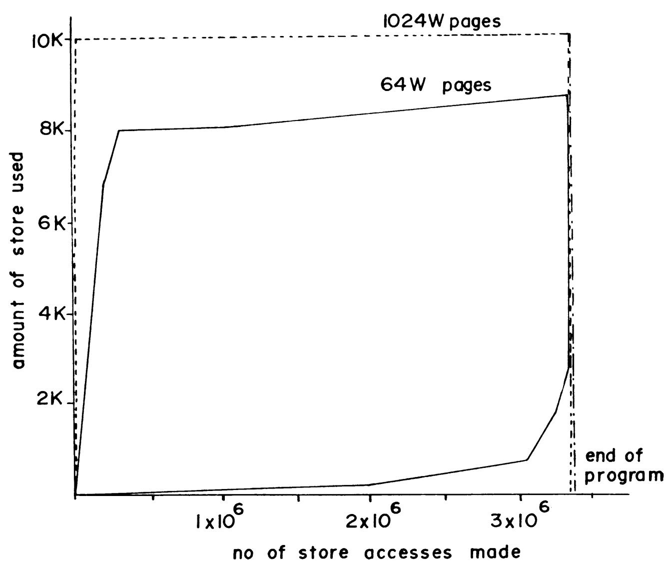

Fig. 4. Variation of store in use with time for a particular program

A second series of simulations was made to relate the way a program accesses its virtual address space with how, in the translation from virtual to main memory space, the total amount of store that it used increased with time. The address sequences were the same as used for the previous set of simulations. Each page in virtual memory was assigned a *pageword* which held such information as the location of the page, its lockout status, the number of accesses made when the page was last in use, etc. It was assumed that the program was stored entirely on the backing store and the appropriate pagewords were updated as pages were brought into use. In order to determine the last use of a page (i.e. when the page could definitely be removed from the main store without needing to be brought back) a separate run of simulations was made in which the address sequences were read backwards so that the first appearance of a reference to a page would be the program's last reference to it under normal running.

Fig. 4 shows the performance of a FORTRAN Compiler (of 8k, and using 2k of working space) under two different page sizes. The upper line, in each case

Downloaded from https://academic.oup.com/comjnl/article/13/1/48/388992 by guest on 19 April 2024

shows the relation between the total amount of store used with time while the lower line shows the way the program relinquishes store. The area between a set of lines, termed the ST factor, represents the product of the store in use by the program and the time for which it is in use; the ST factor can therefore be used to compare the usefulness of different page sizes. It is notable that the slope of the initial steep gradient, in the first phase, is less with small pages, indicating that the rate of information transfer is also smaller in the first phase but this is, of course, obtained by more transfers (the smoothed out lines do not show individual page transfers because of the scale of the graph). The two phases are seen here quite clearly and were found to be present with roughly the same characteristics in other programs analysed; the graph is in many respects similar to one published by Fine *et al.* (1966) for different types of program. They report a large increase of store use between about 2mS and 100mS on the AN/FSQ–32 using 1024w pages and the results obtained here would also indicate that the large growth occurs in the first 100mS (about 40,000 store accesses) for 1024w pages and about 800mS for 64w pages; i.e., that the rate of information transfer required would be, in terms of the actual computation time of the program, 102·4 words/mS of computation time for 1024w pages and 10 words/mS with 64w pages. This information would be demanded by page faults at the rate of about 0·1 faults/mS with 1024w pages and 6·1 faults/mS with 64w pages. [The use of time units in mS has been made here purely for convenience of comparison with the mentioned results: subsequent references to 'time' will be in terms of the number of store accesses made as it is felt that this would make the results more machine-speed independent.] If the ST factor of store in use (i.e. the area between the curves in Fig. 4) is plotted against page size, it is seen **(Fig. 5)** that the initial large saving with small pages is reduced when considering long runs.

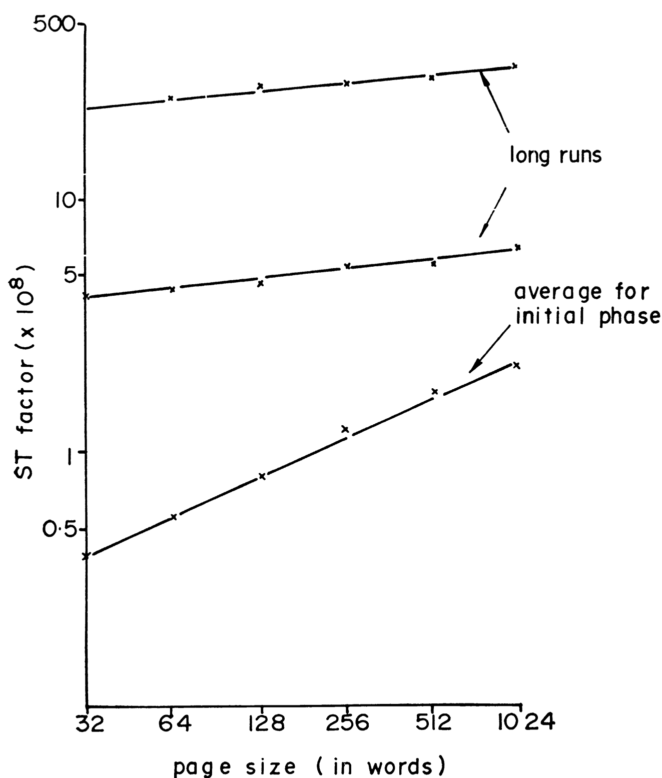

Fig. 5. Increase in ST Factor with page size

These results would suggest that with a large page size of the order of 1024w, the program demands access to a large proportion of its working space in a short interval of time, and the large ST factor of the store in current use would indicate that it does not finally dispense with the use of pages until near the end of its run. This is supported by the published data on the heavy page traffic caused by attempting to restrict the active size of the program to anything less than about two-thirds of its address space when 1024w pages are used. In fact, in order to cause a program to average out its demands for new pages over a longer length of time, it is necessary to use smaller pages and this involves the expense of repeated attention by the supervisor in providing more pages, more often, but with a saving in the amount of store used. As it was seen that the demands of a program are for several small and separate areas in its address space, the greater resolution with small pages would go towards providing this without an excessive use of store space.

The short runs that were analysed here used an average of 8k of store: the longer runs used 10k and, 14k respectively (all these figures are for 1024w pages). It is possible that paging characteristics would be slightly different for very large programs (i.e. larger than 20k) though some published results (Fine *et al.* (1966), Varian and Coffman (1967)) would tend to indicate that this is not so and that most programs, whatever their size, would require over two-thirds of the total number of 1024w pages that they could use. Hence increasing the page size would only tend to increase the amount of store needed though it would reduce the number of new page demands if, at the same time, the program was allowed to load and keep most of its pages in the main store.

5. Prediction of page requests

The expense of having small pages using the demand algorithm is the repeated attention required by the Supervisor and it is clear that efforts will need to be taken to reduce this before a small page size becomes a practical possibility. A study was therefore made of alternative algorithms using a measure of prediction. On first inspection, a form of prediction in which a block adjacent to the currently addressed block is brought into the foreground store may appear to be little different from doubling the page size. However, it was mentioned in the section on the areas of activity that the simulation program gathered data about the number of times a new page was adjacent to one present in the store and this evidence tended to indicate that, even with a small amount of incorrect prediction, the total store used (i.e. the sum of the store actually accessed and that brought in predictively) may be less than that used by the same program in a system where the page was of double the size and where pages were loaded on demand.

Let W be the total amount of store accessed by a program at a particular instant without any predictive loading. If we assume that the available address space is infinite and that the areas within this in use are always

Downloaded from https://academic.oup.com/comjnl/article/13/1/48/388992 by guest on 19 April 2024

separate, then the effect of predictive loading with the program's next new page demand will be to increase the store used to $(W + P) + P$, where P is the page size. If the prediction is successful X times repeatedly, the store used will be $(W + P) + XP$ since there will always be one extra predicted block not accessed by the program. If the prediction fails Y times repeatedly, the store used will be $(W + YP) + YP$ since for every new page demand there will be one unsuccessful prediction.

If the probability of successful prediction is b, then after L new page demands by the program we have

$$\text{Store used} = W + LP + L(1 - b)P + P$$

or

$$\text{Store used} = W + (2L + 1)P - LbP.$$

In the ideal case where $b = 1$ it is seen that with prediction there is always a slight increase in store used; if $b = 1$ and if the same program would use $L/2$ pages with the demand algorithm and twice the page size, the extra store above this used by prediction with the smaller block size would still only be P. The number of program halts without prediction would be L; with prediction this is reduced to $(1 - b)L$ at an expense, in extra store used, of $LP(1 - b) + P$ and if $b = 1$, we find we have reduced program halts to zero. In practice, this expression would be modified by the fact that programs do not tend to use entirely separate areas in their address space so that, after a stage, some areas would tend to run into each other: if, at the same time, replacement routines were paging out parts of the address space, the expression would be further altered.

One method of reducing the increase in store used by faulty predictions, $LP(1 - b) + P$, would be to keep a one block buffer to hold the predicted page. Accordingly, the first algorithm, called One Block Lookahead (OBL), used this principle: whenever the program demanded a new page, R, the page following this, $(R + 1)$, was loaded into a buffer (provided the page was not already in the main store). This was kept locked out and if the program next demanded page $(R + 1)$, the lockout was removed to give immediate access; further, a new prediction was made and page $(R + 2)$ was loaded into a buffer. If the prediction had failed and page T was demanded, then page $(T + 1)$ would be loaded into the buffer as a prediction. In this way a program would never use more than one page above its demanded space (i.e. $LP(1 - b) = 0$) and its total store usage with prediction would be well below that used with the demand algorithm and a page twice as large. The reason for locking out the predicted page is that by this it is possible to have the program drive the prediction algorithm by informing the supervisor exactly when prediction had succeeded.

The second algorithm, called Simple Prediction (SP), was more extravagant in its use of store: whenever the program demanded a new page R, the next page, $(R + 1)$, was also loaded and locked out (as for OBL), and this page was not overwritten even if the next demand was for a new page, T. In this case, page $(T + 1)$ would be loaded with page T so that there would be two predicted pages, $(R + 1)$ and $(T + 1)$, in store. That is, the failed predictions were not overwritten and the program had the opportunity to access predicted pages in any order. This has the advantage of permitting successful prediction even when the program's address space is 'growing' in more than one direction. In a matrix multiplication program, for example, calls may be made alternately to elements of each matrix and while in OBL this could result in the buffer being repeatedly overwritten and prediction failing, in SP both predicted blocks would be available for use at the cost of the increased ST Factor caused by predicting each block too early.

Fig. 6 shows the number of program halts with the demand, OBL and SP algorithms for different sizes of page. It is seen that even with the small increase in store used that is the feature of OBL, there is a significant reduction of between 25% and 35% in program halts while with the SP algorithm the reduction is between 50% and 70% (in this case it may be noticed that there are fewer halts with 32w pages than with 128w pages under the demand algorithm). This reduction is at the expense of the store used and **Fig. 7** shows the ST Factors for programs running under the different algorithms. The increase in ST Factor over that used under the demand algorithm is between 1% and 15% in the case of OBL, and 20% and 30%, in the case of SP. In the latter case the increase brings the ST Factor to almost the same value as for a page twice the size under the demand algorithm.

To measure the effects of using an algorithm somewhere between OBL and SP, a restricted version of SP was used in a more detailed simulation in which only four predicted pages were held in store (see Joseph (1968), p. 81). Here realistic computation and transfer times were simulated and even with the restriction on prediction, backing store delays were reduced by about 2%–12%.

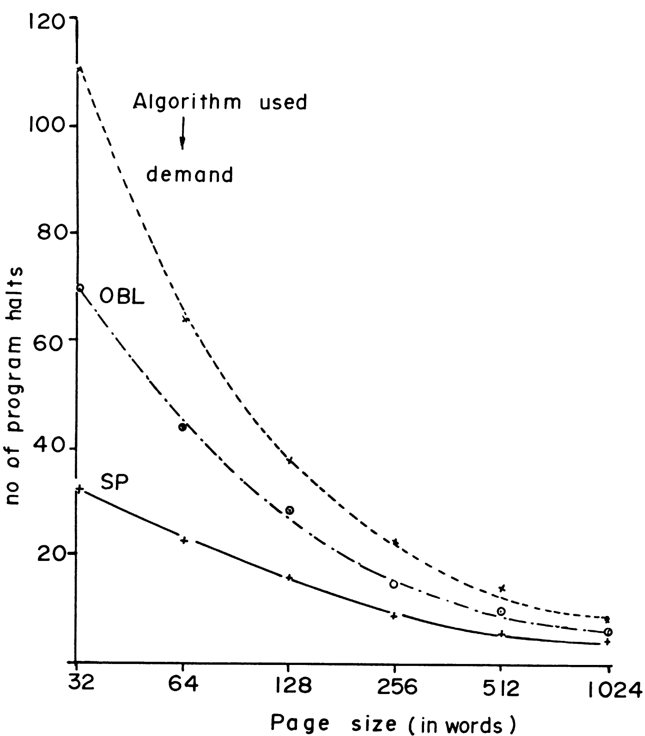

Fig. 6. Number of program halts with three paging-in algorithms

Downloaded from https://academic.oup.com/comjnl/article/13/1/48/388992 by guest on 19 April 2024

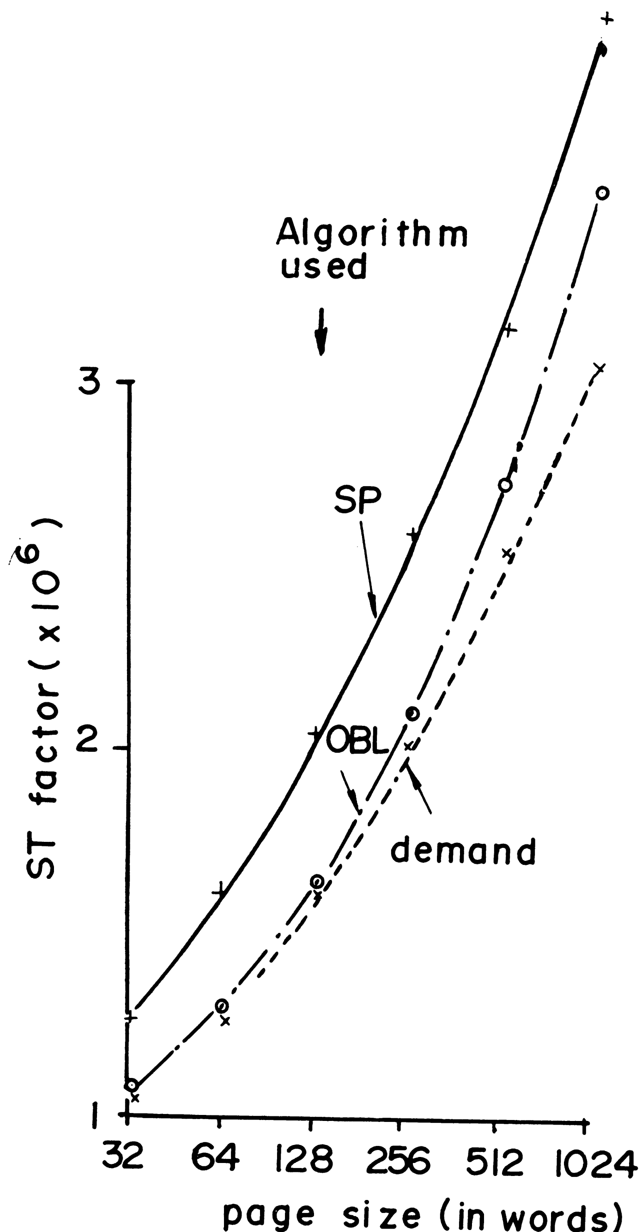

Fig. 7. ST Factor with three paging-in algorithms

One side effect of using predictive algorithms is that the number of backing store transfers necessary to run a program increases with the number of failed predictions. In the case of OBL, the percentage increase was found to vary around 50% and in the case of SP it was about 20%.

6. Conclusions

It is slightly confusing to talk of blocks of stores as 'pages' if their sizes are down to 32w as an implementation with the degree of indirection and relocatability normally associated with that term may prove extremely difficult. A possible method of implementation may be to use a larger size nominally, but to subdivide this into smaller blocks and to use 3 or 4 bits in the page register (or page table entry) to specify the number of these blocks that have been loaded into the main store (this is similar to a proposal by D. J. Wheeler in another connection for a modification to ATLAS 2: see Wheeler (1965)). There will be the possibility of a slight inefficiency as the extra bits can only specify the size of the loaded page, and the effect of loading the first and last blocks in a page will be to consider the whole page in use. But since prediction, in the algorithms tested above, uses the adjacency in virtual address space of the predicted block to the demanded block, this situation may not arise often.

However, even with this measure of success, it is clear that prediction is useful only when the rate of increase of store by a program is large (i.e. in the initial phase, or on the initiation of a new process) and when there is spare backing store channel time. It might then be possible, and indeed desirable, to use prediction in the initial phase to reduce program halts and a suitable replacement algorithm in the second phase to reduce the amount of store used as the advantages in using a small page size (or small blocks within a page) seem to be sufficient to warrant a slight increase in supervisor computation time to serve the need for efficiency.

7. Acknowledgments

This paper is taken from a Ph.D. Thesis submitted while the author was at the University Mathematical Laboratory, Cambridge: the author therefore gratefully acknowledges the advice and encouragement of his Supervisor, Dr. D. J. Wheeler. The author would also like to thank Mr. N. E. Wiseman for his interest and Dr. J. H. Tucker for providing the address sequences used in the simulations. The author was supported for part of the period of this research by an IBM research grant.

Appendix

The address sequences used in the simulations were produced by running programs under an interpretive trace program written by J. H. Tucker. Programs were selected by sampling the multiprogramming workload during the computing service sessions on the University Mathematical Laboratory's Titan computer: these programs were then run under the interpreter and the sequences of addresses collected on magnetic tapes. In all, 18 programs were traced up to either the point where they stopped or where they generated 50,000 addresses, and 2 programs were traced up to about 3·3 million addresses and 820,000 addresses respectively. 6 of the programs were in IIT (The Titan assembly language) and the remainder in Titan Autocode, performing such operations as Runge-Kutta Integration, Data Reduction, Linear Programming etc.

The simulations performed with these address sequences would, to some extent, be affected by the order code

Downloaded from https://academic.oup.com/comjnl/article/13/1/48/388992 by guest on 19 April 2024

of Titan and by the large number of index registers available (Titan was the prototype of ATLAS 2 and so incorporates many of the characteristic ATLAS features). In general, the order code is somewhat distended with fixed length instructions of 48 bits: hence a larger number of store fetches would be required than in machines with a variable length and compact order code. Another feature is the Extracode orders, which are system routines performing 'macro' operations in a privileged mode, and extensive use of Extracodes, whose execution cannot be traced, would affect the generality of the address sequences.

While it is not possible to state exactly how far the results based on these address sequences are machine-dependent, it is likely that the trends in paging exhibited here would be applicable for most machines: this claim would be supported by the agreement of some of the results with others produced on greatly different machines (e.g., the AN/FSQ–32 at SDC). They would be subject to the general limitation that in all programs written for non-paged environments no particular effort is made to contain instruction or data areas in fixed length localities, so that programs written consciously for paged machines should show improved characteristics.

References

BELADY, L. A. (1966). A Study of Replacement Algorithms, *IBM Systems Journal*, Vol. 5, No. 2.

FINE, G. H., MCISAAC, P. V., and JACKSON, C. W. (1966). Dynamic Program Behaviour under Paging, Proceedings 21st National Conference, Assoc. Comp. Mech.

GIBSON, D. H. (1966). Considerations in the Block-Oriented Systems Design, IBM Systems Development Division, Document TR00–1510.

JOSEPH, M. (1968). An Analysis of Storage Hierarchies in Digital Computers, Ph.D. Thesis, University of Cambridge.

KILBURN, T., EDWARDS, D., LANIGAN, M., and SUMNER, F. (1962). One level Storage System, *IRE TRANS. Electronic Computers*, Vol. EC–11, No. 2.

LAUER, H. C. (1967). Bulk Core in a 360/67 Time Sharing System, Proc. FJCC.

NIELSEN, N. R. (1967). The Simulation of Time Sharing Systems, *CACM*, Vol. 10, No. 7.

SCHERR, A. L. (1966). An Analysis of Storage Performance and Dynamic Relocation Techniques, IBM Systems Development Division, Document TR00–1494.

VARIAN, L. C., and COFFMAN, E. G. (1967). An Empirical Study of the Behaviour of Programs in a Paging Environment, Assoc. Comp. Mach. Symposium on Operating System Principles, Gatlinburg, Tennessee.

WHEELER, D. J. (1965). Changes to ATLAS 2 needed for On-Line Working, University Mathematical Laboratory, Cambridge

Book Review

Matrix Analysis of Discontinuous Control Systems, by P. V. Bromberg, 1969; 265 pages (*Macdonald and Co. Ltd.* £5.)

This book is concerned with the application of matrix techniques to the solution of problems involving discontinuous control systems. It is intended for graduate and research students in control engineering and related disciplines. The book consists of seven chapters, the first of which introduces the concepts of control systems, illustrating the ideas by examples from the field of aircraft engineering. In the second chapter, the author gives a résumé of the matrix analysis relevant to the analysis of control systems. The succeeding chapter considers the stability of a motion which is defined by difference, as opposed to differential, equations. Several general theorems based on the method of Lyapunov as applied to the discrete system are given and interpreted in terms of the eigenvalues of certain matrices. In the fourth chapter, it is shown how the results of the third chapter, together with the techniques of chapter two, can be applied to determining the behaviour of discontinuous control systems. Again, several examples in the field of aircraft engineering are included. Chapters five and six continue the application of the earlier theory to relay-operated control systems, and to relay systems subject to external disturbances. The final chapter introduces an extension of the theory to a more general class of problem. The mathematics in this book is relatively easy to follow and the reader is left with no doubt that the book is designed for engineering research workers. The book, a translation from Russian, unfortunately suffers from a great many errors which can only lead to confusion. A common error is the omission of the "dot" to denote differentiation. More seriously, the theorem quoted on page 72 is certainly in error, the word 'stable' unfortunately having replaced the word unstable! Errors of these types make the book annoyingly difficult to follow.

A. R. GOURLAY (Dundee)

Downloaded from https://academic.oup.com/comjnl/article/13/1/48/388992 by guest on 19 April 2024