Demand paging in perspective

by C. J. KUEHNER and B. RANDELL

International Business Machines Corporation
Yorktown Heights, New York

INTRODUCTION

The method of storage allocation known as “demand paging,” first introduced by the designers of the Atlas computer, has a very appealing conceptual simplicity. Furthermore, the possibility that this technique could be used to allow programmers to ignore the problems of organizing information flow between working storage and one or more levels of backing storage was, to say the least, very tempting.

However demand paging has its limitations. Some were foreseen—several years ago Dennis and Glaser pointed out that page-turning was potentially disastrous if applied to the wrong class of information. This however did not prevent the implementing of systems which in at least their initial versions have had grave performance problems arising from excessive paging. Such problems have caused much work on various more-or-less ad hoc attempts to improve system performance. The purpose of this paper is to survey the various techniques that have been proposed or used in an attempt to improve the performance of demand paging systems. By this means it is hoped to make clear the relationships between the various apparently completely separate techniques, and to aid the understanding of both the potentials and the problems of demand paging.

Demand paging

The two essential characteristics of demand paging are clearly conveyed by its title:

(i) Information is demanded by a program at the moment it is needed, without any prior warning. The “demand” is in fact implicit, and arises out of an attempt to use information not currently in working storage.

(ii) Information is transferred to and from working storage in units of a page. Such pages are of equal size (typically 1024 words). Correspondingly, the working storage is regarded as being logically equi-partitioned into page frames.

These characteristics together imply one of the benefits of demand paging, i.e., the removal from the programmer of the burden of explicitly stating what information is to be transferred to and from working storage, and when these transfers are to take place. However, as should become clear in the main part of this paper, it is also these two characteristics which are the source of the performance problems actually experienced in some systems, and predicted by several analytic and simulation studies.

The typical symptom of these performance problems is a low CPU utilization, together with a high utilization of the channels between backing and working storage, arising from an excessive amount of page-turning (excessive, that is, in relation to the amount of processing achieved). A system exhibiting such behavior is often said to be “paging itself to death.”

These inefficiencies derive from two fundamental causes. Firstly a program, while awaiting satisfaction of a page demand, will continue to consume system resources—notably working storage. This is discussed in reference and illustrated graphically in Figure 1. It is clear that if page transfers are very frequent, or consume a lot of time, much of the resource utilization will be non-productive. The second cause is somewhat more subtle, and is due to the non-linear relationship which has been found to hold between program efficiency (e.g. Q average number of instructions executed between page demands), and the number of page frames allocated to the program. This relationship is illustrated in Figure 2. It will be seen that although a program will continue to run quite well with a somewhat smaller set of pages than the totality it references, its performance falls off rapidly when the set of pages is reduced beyond a certain point, often termed the parachor of the program. If a system is such that programs are often forced to operate with less than their parachor of pages in working storage, the system performance is likely to be far from acceptable.
Three basic factors are involved in determining whether a demand paging system will succumb to the fate of being paged to death. These are:

(i) The system hardware characteristics
(ii) The program load
(iii) The operating system strategies

Of these factors the first is in some senses the most important, because an attack based on this area is by far the most likely to be successful in reducing the paging rate to an acceptable level. For example a drastic increase in the size of the working storage will overcome all but the most perverse of operating system strategies. However, adding more working storage capacity indefinitely is of course a trivial solution. The difficult question is what is the minimum amount of working storage which must be added in combination with various backing store designs to produce the required level of performance at minimum cost? Nielsen \(^5\) has shown the effect of unbalanced system designs in which optimization of any but the critical system element has had little effect on overall system performance. A description of techniques available for determining the components of this balance in specific systems is beyond the scope of this paper,\(^5,9\) however several general remarks may be in order.

Belady \(^7\) has shown the effect upon system performance of reducing the delay associated with obtaining information from backing stores. From such an analysis it is clear that as one reduces the delay in obtaining a required page, one can obtain the same performance level with a significantly smaller amount of available working storage. Such results suggest the use of bulk core store as the solution to the performance problem.\(^9\) However, at this time it is difficult to justify the use of such devices on a cost/performance basis in any but the most demanding of circumstances due to their relatively high cost. Conversely, the same results may be interpreted to show that flailing arm disks are a poor choice as backing storage in demand-paging environments. What effectively occurs in systems employing these devices is that the core latency effect shown in Figure 1 prohibits efficient operation unless a vast amount of working storage capacity is available. Consequently even though such devices are cheap, the performance obtainable does not yield a viable cost/performance ratio.

If one is free to vary the hardware complement, it can almost be guaranteed that an acceptable level of performance may be obtained. Such hardware solutions, however, are in some senses a brute force approach aimed at providing an environment which is insensitive to the difficulties caused by program load and operating system strategies. On the other hand, attacks on the factors of program load and operating system strategies can be considered as attempts to extend the area of practicality of demand paging. Therefore the remainder of this paper is concerned solely with the various proposed software solutions to the problems of demand paging.

Prior to entering into this discussion however, two notes of caution should be injected. First is the question of cost. The cost of hardware solutions is easily de-
terminable in terms of rental or purchase dollars. This is not the case with software solutions in general. Although one would like to compare specific solutions of equal cost, it is not possible without a knowledge of the cost and skill of the software talent available to a specific installation. Second, the implementation of one, two or all of the techniques described below does not guarantee the transformation from an unacceptable to an acceptable level of system performance. In fact, a priori, it is very difficult to predict the magnitude of the performance improvement which would be obtained on a specific system by implementing one or several of these techniques.

**Program load**

The instructions and data whose storage and transmission form the load on the system resources such as storage and channels are comprised of both user programs and system programs. In fact certain parts of the operating system are users of the very resources that they have the task of allocating.

This load has properties of both volume and structure, both of which can be the source of problems. The effect of the volume of data and instructions on limited storage and channel resources should be readily understandable. The effects due to the structure of the program load are much more subtle. There is one easily-determined measure relating to the structure of a given program load, namely the working set size. The working set of a program is the collection of distinct pages to which storage references were made during a given execution interval. The largest working set size of a program is then the number of pages occupied by the program and its data. At the opposite end of the spectrum a single page is also a working set, but for an extremely short execution interval. (To complete the execution of even a single instruction will normally require more than one page.) Working set size is hence a function of the interval during which storage references are observed. Obviously a plot of average working set size against length of execution interval observed is monotonically increasing. However, it should be remembered that for a given interval of observation different stages during execution of a program may well result in extremely different working set sizes and compositions.

Unfortunately the working set size measure is far abstracted from the underlying characteristics of program load structure which affect its value. These characteristics include:

(i) programming style
(ii) degree of modularity of code
(iii) layout of data

The various techniques for increasing the efficiency of paging systems described in this section all have as their goal a reduction primarily in the average value of the size of the working set and also in the volume of the program load. It is convenient to discuss separately techniques which can be supplied to an existing program load, and those of relevance during the initial design of programs for a paging system.

**Program load improvement without recoding**

The prospect of modifying the instruction flow and data layouts of an existing complex program load in order to reduce the average working set size is not exactly attractive. Hence some attempts have been made to achieve such a reduction by simply repacking the modules (which may be greater or less than a page in extent) that make up the program layout in virtual memory. Comeau has reported on a brief series of experiments in which the modules which constituted an entire operating system were presented to the system loader in various different orders. One rearrangement was found that reduced the rate of page transfers by over 80% from that caused by the original version of the operating system.

This technique is obviously very simple to carry out, assuming one knows how to choose a good reordering of the modules. Any rational choice requires at least some information about the dynamic interaction of program modules and the pattern of referencing activity. Possible classes of information include:

(i) frequency of reference to the various modules
(ii) sequence of references to the modules
(iii) frequency of reference to the various pages
(iv) sequence of references to pages

Such information could be obtained during trial runs of the operating system. Two problems immediately come to mind. Firstly, there is no guarantee of consistency in data gathered from different runs. Secondly, the data regarding modules concerned with the actual paging function will vary as the working set composition changes, due to different packing arrangements. However these are probably minor problems compared with that of computing an optimal, or even near-optimal module ordering from the obtainable reference information.

Summing up, in the present state-of-the-art, any but the most minor attempts at re-packing are probably best regarded as last-ditch efforts at recovering from inadequate hardware, operating system strategies, and/or programming style.

**Program load improvement via redesign and recoding**

The style of program design required to obtain good
Adherence to the rules must be maintained during program modification (in the case of extensive and repeated improve their paging performance. For example the modification this can be quite difficult). Therefore of necessity systems programmers must be particularly circumspect. It should also be noted that adherence, by both programmers and compilers, to a set of programming commandments is as important as it is distasteful. The distastefulness arises from the need for a continual awareness of the part of the programmer of the (at least approximate) position of page boundaries in relation to his instruction and data layouts.

Necessary commandments include:

(i) Do not reference a wide variety of pages in rapid succession. It is better to localize activity for a reasonable interval, and then move on to a different locality, rather than to intermix references to the much larger combined locality.

(ii) Minimize space requirements for instruction and data storage only insofar as this permits adherence to commandment (i). (The assumption here is that it is worth trading increased backing storage utilization for decreased paging activity).

(iii) Avoid excessive modularity of programs. Program modularity, although very helpful if one must introduce changes and modifications to existing codes, can reduce execution performance substantially. If an operating system is composed of literally hundreds of program modules, any but the most trivial of work requests necessitates tens or hundreds of control transfers. Such transfers are costly in terms of unproductive CPU time but even more critically, in the worst case, each transfer could require that a distinct page of information be referenced. It should be clear, therefore, that a distinct trade-off exists between the effective static level of program modularity and the amount of dynamic control transfers which directly result.

(iv) Design data layouts to take account of the order in which data items are to be processed. For example, if an nxn matrix (where n is slightly larger than the page size) is to be scanned repeatedly by rows then storing it by columns would be unwise, to say the least. (For a detailed discussion see McKellar and Coffman.13)

The more frequently a particular program is to be used, the more important is adherence to these rules. Therefore of necessity programmers must be particularly circumspect. It should also be noted that adherence to the rules must be maintained during program modification (in the case of extensive and repeated modification this can be quite difficult).

However there have been some quite successful attempts to redesign fairly simple algorithms in order to improve their paging performance. For example the papers by Cohen14 and by Bobrow and Murphy15 describe list processing systems in which the dynamic storage allocation of list elements has been designed to group elements which are likely to be referenced in quick succession into the same page. Each system has thus enabled very large list processing applications to be run, with an efficiency surprisingly close to that attained by small applications which fit entirely within core storage (for example Cohen reports that only a factor of 3 in speed was lost, even though the average access to backing store was 104 slower than access to core storage).

Another study which gives evidence of the performance improvements that can be obtained, particularly when a program has a very limited number of page frames allocated to it, is that by Brawn and Gustavson.16 This report describes an extensive set of experiments investigating the effect of programming style on performance obtainable from the M44/44X experimental demand paging system. The experiments involving improvement of a sorting algorithm are particularly interesting. A series of changes were made to the original straightforward implementation of the algorithm. Each change affected the sequence, but not the number, of page references. The changes were simple to implement, but involved considerable thought about the logic of the program and its behavior in a paging environment. It was found that the amount of real core storage needed for reasonable performance was reduced by a factor of over six.

Operating system strategies

If the hardware and the program load are such that programs can usually have more than their parchor of pages in working storage while they are executing, then quite simple operating system strategies will suffice. However in general very careful thought must be given to the problem of achieving a high probability that there is always a task in working storage ready for CPU activity, and avoiding running programs in an excessively space-squeezed environment. This is difficult enough in a simple multiprogramming environment, but is even worse in a time sharing environment. In such circumstances there is the added need to give all users frequent service, so that simple requests can be answered quickly. Hence execution requirements which turn out to be lengthy will be subject to time slicing and, in all probability, to being paged in and out of working storage repeatedly.

It is convenient to consider the functions of an operating system which are relevant to this paper as being scheduling, allocation, and dispatching. Here scheduling is understood to be the maintenance of an ordered list of the jobs that are competing for the ser-
services of the allocator. The allocator controls the allocation of working storage between such jobs from this list as it chooses to service. Finally, should there be more than one job ready and waiting for a CPU when one becomes available, the dispatcher makes the choice. Ideally a design should guarantee the harmonious cooperation of these three mechanisms under a wide variety of load conditions. Unfortunately even untried theoretical approaches to this goal are few in number, one of the most developed being that of Denning.11

However we are getting beyond the scope of this paper. The rest of this section is a discussion of various more or less isolated approaches to improving the strategies, and hopefully the performance, of demand paging systems. The techniques to be described can be divided into those concerned primarily with scheduling and those concerned with allocation.

Scheduling

In paging systems experiencing performance problems, the scheduling function must be viewed in a somewhat different light than one normally finds in the literature, where scheduling algorithms are compared and optimized in isolation of the system being scheduled. In such cases scheduling is considered the dominant system controlling function and allocation is reduced to providing space for jobs as the scheduler dictates. The result of such an approach in a system containing a resource class more critical than processing time is, in general, a significant degradation of performance. The problem is that few “schedulers” sample the resources of the system and base their decisions on the results. Instead they attempt to inject and remove jobs from execution on the basis of arbitrary pre-set bounds for time-slice, quantum time, amount of occupied storage, etc. The effect upon performance in a demand-paging system of cycling a job into and out of execution is critical. One can easily see why this may be so from an analysis of Figure 1.

The key to improving performance via scheduling in such systems is to subordinate the scheduling function to that of allocation. Jobs should be introduced and removed from the system on the basis of the state of the system resources by the scheduler under the control of the allocator (with few if any exceptions). Once a job has been introduced into the system and acquires its parachor of pages, one should be hesitant to remove it and lose the investment unless it has been in execution for a lengthy period of time. The period of time-slice should be an upper bound exception and not the general rule of operation.

The above philosophy of scheduling is concerned with when, and under whose control, items should be entered or removed from the list of jobs competing for working storage. There remains the problem of which jobs should be selected for adding to the list when circumstances make this desirable. The two scheduling techniques that have been proposed for improving the performance of demand paging systems, which are described below, are concerned with this latter problem area.

A fairly simple and frequently proposed technique of reducing the amount of paging is to arrange that requests for use of the same program are batched together. This can be quite successful, particularly in those time-sharing systems where many users are making continual use of a substantial program, such as a compiler, or a text editing and filing program. In a very large system the fact that the similar requests are batched together might not even be apparent to the user, the delays caused by waiting for a batch to be accumulated being quite small and perhaps even being outweighed by the effect that reducing the amount of paging has on system efficiency.

A more autocratic proposal12 is to dynamically limit the programmer response rate. The theory is that although a system should respond to a programmer’s request immediately, it will be good for the system, and in all probability the programmer, if his rate of response to the system is controlled. One simple method would be to lock the terminal keyboard after a system response, for a period of time in some way proportional to the amount of system resources used in providing the response. This is intended both to limit the load in the system, and to encourage users to think before they type. The acceptability or quantitative performance improvement that such a scheme would yield remains to be proven.

Allocation

As mentioned earlier, it is considered the function of the allocator to decide when to allow the scheduler to introduce new jobs into contention for working storage space and to decide which requests for working storage it should attempt to service. This enables the allocator to control the level of multiprogramming (i.e., number of independent contenders) in working storage during any time interval. If the modifications to the existing operating system modules would be too extensive to incorporate this scheduling-allocation procedure, one can introduce a new almost autonomous mechanism to the original operating system to perform such a function. Such a mechanism, termed the “load-leveler,” was in fact added to the M44/44X system. The scheme chosen for the load leveler was to periodically examine recent CPU utilization and paging rate, and when necessary request the scheduler to temporarily remove jobs from the list of those being serviced by the allocator.
The jobs which are temporarily set aside are chosen from among those which are not experiencing frequent interaction.

The need to move a job which has reached the end of its time slice out of working storage temporarily can be a source of considerable difficulty. Such a job may have painstakingly accumulated the ration of page frames that it needs for effective progress. If on next being eligible for a time slice it must demand pages one-at-a-time, its initial progress will be minimal. (This is borne out by the simulation results given by Fine et al.) The technique known as "pre-paging" (Oppenheimer and Weizer) is intended to avoid this situation. Pre-paging involves fetching a set of pages, which are believed to be the working set of a job, into working storage before the job is allocated to a CPU. The obvious difficulty with pre-paging is deciding what to pre-page (i.e., the composition of the working set). The problem which involves predicting future program activity, is very similar to that of designing a page replacement algorithm. A straightforward approach is to pre-page the last \( n \) pages referenced during the program's previous time slice. A major factor in choosing the magnitude of \( n \) should be the speed of the backing store. One theory is that the longer it takes to fetch a page on demand, the larger \( n \) should be. Although this will increase the probability of fetching unwanted pages, it should avoid many subsequent page demands and reduce the overall space-time product for the execution of this job.

A logical extension of pre-paging is "paging by function." This involves the use of special program directives indicating that a certain set of pages will always be required for a certain function to be performed. These pages will then all be brought to working storage at once, rather than one by one, on demand. A corresponding directive can be used as a form of "block page release," indicating that all of the indicated pages have been finished with, and can be replaced. A possible source of the information needed to use these program directives intelligently might be the sort of program module interaction data discussed in an earlier section.

Lessons to be learned

It is convenient to attempt to summarize the above discussion by considering separately the effects of the demand and the paging concepts.

The idea of having a running program notify the system of its need for a particular resource when it has reached the state of not being able to make any further progress until that resource is granted to it is at one end of the spectrum of possible resource allocation strategies. The other extreme is to allocate the entire resource requirement to a program from the beginning of its life time in the system. (This of course requires complete knowledge of all the resources that a program will need.) In the former case inefficiencies arise from non-productive resource utilization while a program is awaiting satisfaction of its demands. In the latter case the inefficiency arises from giving a program resources for a longer period than it requires thereby making them unavailable to other users. Corresponding to the above comments on resource allocation, a similar discussion applies to the spectrum of strategies possible for controlling the deallocation of resources.

A correct choice of allocation and deallocation strategies for a particular system involves an appreciation of the cost of non-productive resource utilization, the time taken to satisfy a resource demand, the frequency with which such demands are likely to be made, and the probability of getting accurate advance notification of resource needs.

It should be noted that this problem is not peculiar to paging systems, or even indeed to the particular resource of storage space. However experience has shown that a wrong choice of strategy can have drastic effects in a dynamic storage allocation system. This would appear to be due to the fact that in most systems working storage is a very critical resource, and also because of the non-linear relationship that holds between program efficiency and space allotment. (Experimental evidence for this relationship comes from paging systems, but there is no reason to believe that it may not also apply to dynamic allocation systems which allocate differing sizes of blocks of working storage.)

On the subject of paging, it is clear that one of the most obvious difficulties that arise in the design of a paging system is that of choosing a page size. From the viewpoint of avoiding the transfer to working storage of a large amount of information which may not be used, a small page size is desirable. On the other hand if too small a page size is chosen the overhead caused by very large page tables will be excessive. It seems clear that if the basic hardware and operating system strategies are adequate for the program load, an adequate choice of page size can be made, and a very simple page replacement technique for working storage allocation will suffice. The problem arises when this is not the case, and it is necessary for programmers to maintain a continual awareness of the underlying paging environment while designing or modifying programs. In these circumstances the page size becomes a straight jacket and, if only for reasons of programmer convenience, a multiplicity of sizes of storage areas would be preferable. There is in fact some evidence to suggest that such an environment permits higher utilization of working storage than one in which only a single page size is permitted. However so much depends on programming
style and methods of compilation that a definitive comparison is extremely difficult.

CONCLUSIONS

The software techniques for improving the performance of demand paging systems described above all attempt in their various ways to perform one or more of the following functions:

(i) provide advance warning of page demands
(ii) reduce the working set of programs, and hence their parachor
(iii) ensure that programs are not excessively space-squeezed (ideally that each program is given just over its parachor of pages)
(iv) provide explicit identification of the working set.

When one attempts to abstract the common denominator of these techniques it becomes clear that they are not just attempts to “tune” the basic demand-paging philosophy, they in fact attempt to negate both the “demand” and the “paging” characteristics. Preparing, for example, attempts to avoid the “demand” page exception with its attendant overhead and delay, as well as avoiding “paging” by attempting to select a meaningful portion of the program to preload. The result is a storage management system which is a mixture of strict paging and slow swapping.

It is important therefore to remember that demand paging is but one of the many possible techniques of dynamic storage allocation. As with any strategy its range of efficient use is limited. The simplicity of working with uniform units of allocation and the potential efficiency of bringing into working storage only those units of information which are actually referenced are the strong points of demand paging. The demand paging systems so far implemented have in general shown that these advantages are purchased at the cost of the strong points of demand paging must however await the accumulation of more experience of their use in actual user environments.

ACKNOWLEDGMENTS

The authors would like to thank L. A. Belady, M. Lehman and F. Zurcher for numerous discussions and suggestions which aided in the preparation of this paper.

REFERENCES

1 T KILBURN  D B G EDWARDS  M J LANIGAN  F H SUMNER
One-level storage system

IRE Transactions on Electronic Computers 11 2 1962 pp 223-235
2 J B DENNIS  E L GLASER
The structure of on-line information processing systems
Proc 2nd Congress on Information Systems Sciences Spartan Books Washington DC 1965 pp 5-14
3 J N BAIRSTOW
Time-sharing
Electronic Design 16 9 1968 pp CI-C22
4 J L SMITH
Multiprogramming under a page on demand strategy
Comm ACM 10 10 1967 pp 636-646
5 G H FINE  C W JACKSON  P V MELSAAC
Dynamic program behavior under paging
6 B RANDELL  C J KUEHNER
Dynamic storage allocation systems
Comm ACM 11 5 1968 pp 297-306
7 L A BELADY  C J KUEHNER
Dynamic space sharing in computer systems
8 N R NIELSEN
The simulation of time sharing systems
Comm ACM 10 7 1967 pp 397-412
9 J E SHEMER  G A SHIPPEY
Statistical analysis of paging and segmented computer systems
10 H C LAUER
Bulk core in a 300/07 time-sharing system
11 P J DENNING
The working set model for program behavior
Comm ACM 11 5 1968 pp 323-333
12 L W COMEAU
A study of the effect of user program optimization in a paging system
ACM Symposium on Operating System Principles Gatlinburg Tennessee Oct 1-4 1967
13 A C McKELLAR  E G COFFMAN
The organization of matrices and matrix operations in a paged multiprogramming environment
14 J A COHEN
Use of fast and slow memories in listing processing languages
Comm ACM 10 2 1967 pp 82-86
15 D G BOBROW  D L MURPHY
Structure of a LISP system using two-level storage
Comm ACM 10 3 1967 pp 155-159
16 B BRAYTON  F GUSTAVSON
An evaluation of program performance on the M44/44X system
17 R L PATRICK
Time-sharing tally sheet
Datamation 13 11 1967 pp 42-47
18 G OPPENHEIMER  N WEIZER
Resource management for a medium scale time sharing operating system
Comm ACM 11 5 1968 pp 313-322
19 L A BELADY

From the collection of the Computer History Museum (www.computerhistory.org)
<table>
<thead>
<tr>
<th>A study of replacement algorithms for a virtual storage computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Systems J 5 2 1966 pp 78-101</td>
</tr>
<tr>
<td>20 B RANDELL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A note on storage fragmentation and program segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report RC 2102 IBM T J Watson Research Center Yorktown</td>
</tr>
<tr>
<td>Heights New York May 1968</td>
</tr>
</tbody>
</table>