Program reference patterns can have a more profound effect on paging performance in a virtual memory system than page replacement algorithms.

This paper describes experimental techniques that can significantly reduce paging exceptions in existing, frequently executed programs. Automated procedures reorder relocatable program sectors, and computer displays of memory usage facilitate further optimization of program structure.

Program restructuring for virtual memory

by D. J. Hatfield and J. Gerald

Experimental techniques have been developed for improving the performance of programs in virtual memory systems[1,2] by rearranging, or by duplicating and rearranging, relocatable sectors of code. Experimental results were obtained from finished programs, rather than from programs in the design stage. Improvements in paging performance were on the order of two-to-one to ten-to-one. Those aspects of the sector placement problem that can be solved without extensive recoding are emphasized. The approach used involves a combination of algorithms for automatic placement of relocatable sectors and examination of memory usage with the aid of a computer display.

virtual memory systems

Virtual memories allow programs to be written as though a virtually unlimited amount of main storage space were available to them. The large virtual storage not only relieves programmers of planning overlay structures for programs larger than available physical main storage space, it makes possible more efficient use of computing system resources. Since programs do not use all of their main storage space at all times during their execution, it is possible to keep only small pieces (pages) in main storage at any given time. Since virtual memory systems provide automatic address translation, additional pages can be brought into physical memory as they are needed and wherever there is room for them. In this way, the pages of many programs can be in main storage at the same time, with the remaining pages kept

on high-speed secondary storage. This increases the possibility of overlapping central processing unit (CPU) and input/output (I/O) operations, improving system efficiency.

Several variables influence the performance of virtual memory systems, including the speed of the secondary storage device, the size of the page, the replacement algorithm for swapping pages, and the structure of the program. Programs written under the assumption that main storage is virtually unlimited can result in a phenomenon called "thrashing," in which much more time is spent performing paging I/O operations than executing instructions. The cost of thrashing is compounded since, as paging I/O increases, the rate of system degradation also increases. For frequently executed application or support programs not written for use in a paging system, some alternatives to complete recoding are clearly desirable.

We present some experimental techniques for examining programs that are to be run in virtual memory systems and for reducing their physical memory requirements with little or no recoding. We assume that pages of contiguous code are relocated as units and that contiguous locations within a page of virtual memory are also contiguous in physical memory. At first we assume that all pages have the same length. This is true of many of the paging (relocate) systems in use today, and it simplifies virtual memory management considerably. Later the model is extended to the case of nested page sizes.

The failure to locate a virtual memory address in physical memory is called a page exception. The hardware environment is generally describable as a memory hierarchy consisting of smaller, faster memories replenished when necessary from larger, slower ones. One goal of the management of such a hierarchy is to minimize the sum of the replenishment times for a given program or programs. Detailed techniques are described in this paper for approaching the problem in the simple two-level heirarchy of main storage and secondary storage, and the extensions necessary for three or more levels are considered briefly.

We first consider the page exception problem and how it relates to page replacement algorithms and to program structure. We then investigate means for representing program intercommunications in matrix form and for performing operations on the matrix to determine a reordering of program parts that will reduce page exceptions. We next examine how additional reordering based on code usage can further reduce page exceptions. Finally, we consider possible extensions to the techniques, including the possibility of compilers that optimize code for execution in a paging environment.

The page exception problem

replacement algorithms

For the purpose of defining a repeatable environment, consider a single program being executed in a physical memory smaller than the span of virtual memory it uses. One would like to minimize the number of page exceptions for a given page size (the page size for virtual and real pages is identical). One way to go about this is to study the effect of the page replacement algorithm,[3] which selects a page to be removed or written over whenever a new virtual memory page must be brought into physical memory. Several comparisons of page replacement strategies have been made, often noting as much as 30 to 40 percent improvement from one algorithm to another. In particular, an algorithm has been found[3] that gives the minimum number of page exceptions for a program. Using this algorithm, one can compare the performance of any other algorithm against the optimal performance.

Although there are good reasons for studying replacement algorithms, the search for an ideal replacement algorithm is complicated by several factors. The minimum replacement algorithm is practically unrealizable, as it requires a knowledge of the future page references of the program every time a page fault occurs. In general, the better a replacement algorithm, the harder it is to implement. For instance, least recently used (LRU) stacks give better performance than a single resettable used bit for each page, but LRU stacks are more expensive to implement in hardware. In addition, the algorithm that overlays the least recently used page, which works comparatively well in most situations, is extremely poor for recurrent cyclic excursions through a large span of main storage. This request pattern can turn up locally in the course of an assembly or compilation, or during matrix multiplication. To make things a little worse, a multiprogramming environment may bring together a spectrum of users with different exception rates and different responses to changes in their available working space. Thus the question of whose least recently used page becomes important unless all users can bring in a new page only on top of one of their own pages (page against themselves).

reducing page exceptions

Another way of reducing the page exceptions associated with a program is to have the program request fewer pages during the course of its execution.[4] For any program, the sequence of virtual page requests is an absolute measure of the page requirements of the program, independently of available physical space or page replacement algorithm. Anything that both reduces the length of the overall sequence and the number of distinct pages used in subsequences should result in reducing the program's page exceptions for almost all physical memory sizes and page replacement algorithms.

Assume that the page reference sequence has been compacted to remove all repetitions of single pages, so that $b\ a\ a \cdots a\ c$ is reduced to $b\ a\ c$. Ideally, the program would have a page of virtual memory removed from physical memory only when there is no more need for the information on that page. As the number of such real pages available to it is decreased, such a program must localize its use of virtual memory to a correspondingly smaller span of pages for longer and longer periods of time.

An obvious first guess as to how to bring about this localization is to take parts of the program that are used closely together in time and put them close together in space. For instance, one would put each instruction next to the instruction or data location it refers to most. Ignoring the fact that this often is impossible, as when instruction q is the most popular reference of instructions r, s, and t, the size of the problem precludes an optimum solution if one exists. The resulting program would not, in the vast majority of cases, be processible. But although individual instructions cannot be moved around at will in virtual memory, larger aggregates of code and data can. Arrays in FORTRAN COMMON can be reordered at compilation time, and subroutines can be reloaded, as can the procedure and data areas of many assemblies and compilations.

An experiment performed in 1967 by L. W. Comeau[5] indicates that the conjunction or disjunction of relocatable sectors of code over virtual pages can have a profound effect on paging performance. Changing the load-time ordering of the modules in a monitor system resulted in a five-to-one reduction in total page exceptions generated during the course of an assembly. The four deck orderings were compared under the same physical memory constraint and the same replacement algorithm. The orderings were: alphabetical (6500 exceptions), random (4200 exceptions), an arrangement based on knowledge of the functions of the modules and awareness of the page size (2400 exceptions), and an arrangement based on knowledge of the modules and a record of paging actions generated while the job was in execution (1200 exceptions). A subsequent experiment confirmed the relative importance of sector ordering over replacement algorithm for this monitor during compilations. The best ordering in Comeau's original experiment was made by a programmer familiar with the functional nearness requirements of the sectors, and aware of the page exception rates of the individual pages. The technique of automatic ordering based on a program's knowledge of nearness requirements is considered now.

Automatic sector reordering

Our aim is to insure that sectors of a program that are needed within a relatively short time of one another are found either in

the same virtual page or in pages that would otherwise tend to be in physical memory at the same time. In order to define the problem, we make certain simplifications. First, the replacement algorithm is ignored except that it is assumed to bring in and to remove one page at a time. Second, the sectors are considered to be opaque, since we are not concerned with what goes on inside each sector. We assume that the average size of a relocatable sector is small with respect to the size of a page (between one-tenth and one-third page size). Ideally then each page should be filled with sectors that communicate more with one another than with any other sectors. In addition, each page should be filled with sectors that are used with nearly the same frequency; we defer consideration of this until later, since it is partly achieved as a result of clustering on the basis of sector communication.

the nearness matrix

Consider a set of m relocatable sectors occupying n pages in virtual memory. Sector i should be near sector j (both should be in physical memory together) whenever control is transferred from an instruction in sector i to one in sector j, or whenever an instruction in one sector refers to data in the other. If all such transfers of control and references are known, a nearness matrix $\mathbf{C}_{ij}$ can be constructed. The value of the element c_{ij} is incremented whenever there is a transfer of control or a data reference from sector i to sector j. This nearness matrix is an $m \times m$ matrix, with one row and one column for each sector i. Corresponding to each different load ordering of the sectors, there is a reordering of both rows and columns of the nearness matrix, and the load orderings that cause the large values in $\mathbf{C}_{ij}$ to be clustered around the diagonal are the "best" orderings. (The diagonal elements themselves represent intrasector communications.)

It is desirable to have a precise evaluator for a given arrangement of the rows and columns in the matrix. The evaluator should be related to the expected number of page exceptions associated with a given sector ordering and a given number r of real page frames in physical memory. (A page frame is the block of main storage into which a page is loaded.) If for the sake of simplicity all sectors are the same fraction of a page size (no sector is loaded across a page boundary), a reasonable evaluator might be the sum

$$\sum_{i,j=1}^{m} c_{ij}\, p_{ij}$$

where c_{ij} is the entry in the nearness matrix and p_{ij} is the probability that sectors i and j are both in physical memory whenever either is in physical memory. If two sectors i and j are in the same page, $p_{ij} = 1$, so that for each page α,

$$\sum_{i,j\in\alpha} c_{ij}$$

is a term in the evaluator. If sector i is in page α and sector j is in page β, then

$$\sum_{\substack{i\epsilon\alpha \\ j\epsilon\beta}} c_{ij}\, p_{\alpha\beta}$$

is a term in the evaluator, where $p_{\alpha\beta}$ is the probability that virtual pages α and β are both in physical memory whenever either is in physical memory. Unfortunately, $p_{\alpha\beta}$ is difficult to estimate, since it depends on the number r of real page frames and the initial state of the physical memory. One needs the probabilities for going from one memory state to another, where a memory state is determined by the r pages in physical memory. To find them is computationally exorbitant when the transition probabilities among pages are known. And until the sectors are assigned to pages, these transition probabilities cannot even be determined. So as a first cut at finding a good ordering, we decided to maximize the sum

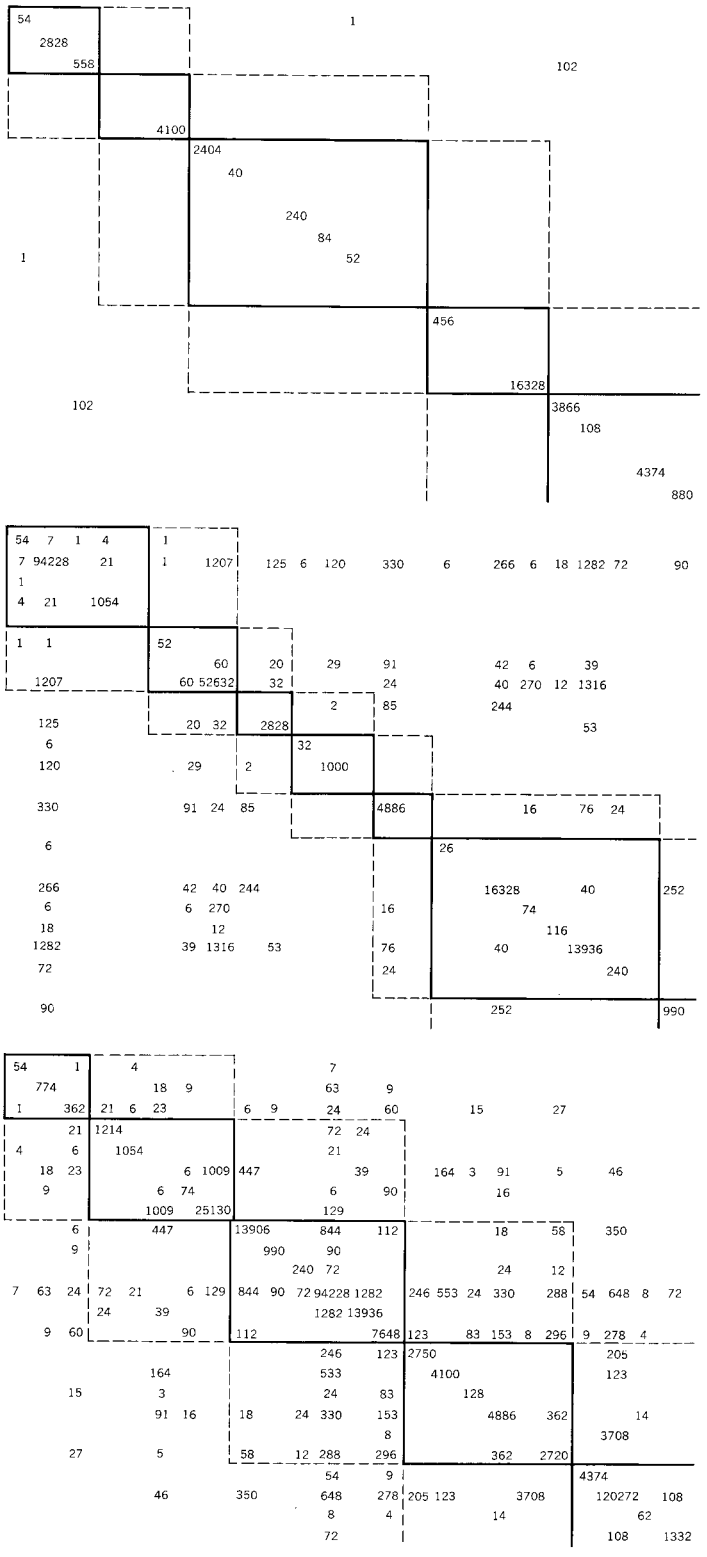
$$\sum_{i,\, j\epsilon\alpha} c_{ij}$$

for the virtual pages α, i.e., maximize the sum of the values c_{ij} for sectors within virtual pages (minimize for sectors across virtual pages). Still assuming that each page contains an exact integral number of sectors, we may associate with each page containing s sectors an $s \times s$ submatrix about the diagonal. A "good" ordering is one that maximizes the values of the nearness matrix in the submatrix positions. Intersector communications for part of a compiler are shown in Figure 1. The square submatrices about the diagonal show the results of a random ordering of sectors at Figure 1A, of the order used by the compiler developers at B, and of an automatic reordering of sectors at C. The dashed lines delimit the number of transfers between adjacent pages; larger numbers here are associated with smaller probabilities of generating page exceptions when using a sector that crosses a page boundary. Since only the first 24 rows and columns of the matrix are shown, the sectors involved are not necessarily the same in all three cases.

trace data

For the programs examined, the nearness matrix was constructed by processing a full instruction trace of the program. All control transfers and data references were recorded. Input/output command chains were not decoded, and I/O data reference lengths were not recorded. Given a list of the load points of sectors, all virtual addresses could be translated into sector references. Control transfers from sector i to sector j incremented c_{ij}, as did an instruction in sector i referring to data in sector j. Instructions involving multiple data references to different sectors were treated as though the instruction had referred to each sector individually.

Figure 1 Matrix reflects three sector orderings



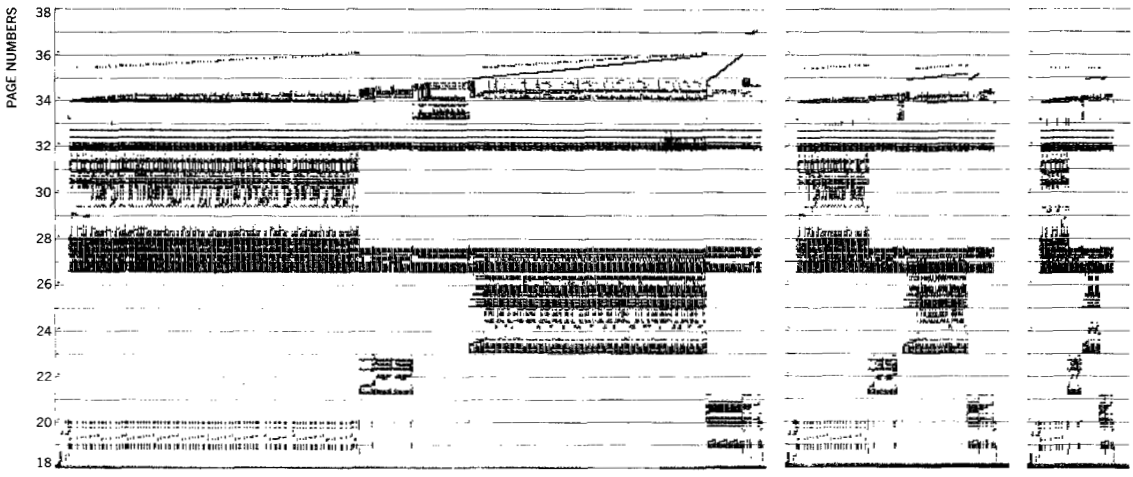
The resulting matrix represents the sum of the program's incremental nearness requirements. As such, it is open to the criticism that it does not adequately represent the more local nearness requirements upon which paging depends. It is also open to the criticism that one program execution only represents its own nearness requirements, and a new nearness matrix must be generated every time that the program is run. The only decisive way to answer these criticisms is by improving the techniques for generating the matrix and by showing that an ordering based on the matrix generated from one program run gives improved performance over a range of runs of the same program using different data. This also presupposes agreement on what subset of data samples comprises a "typical" set, and it is difficult to find general agreement on what data is typical for any program.

The various local phases of the program may be biased in as simple or complicated a manner as is desired. First, it is obvious that whenever a new overlay of program or data sectors is made, a new matrix must be started. And if desired, new matrices may be generated periodically on completing a certain number of instructions, transfers, data references, or additions to c_{ij}; and the individual matrices may be weighted by some function of the virtual memory space traversed. The space traversed between two successive additions to c_{ij} is a measure of the tendency of two sectors to cause a page exception if they are needed together and are not in the same page. It is difficult to quantify this tendency.

Detailed examination of page exception rates as a function of available real space does not yield few or simple expressions. Specifically, the curves are often not well approximated by an exponential or simple algebraic function of real space. In other words, a program does not necessarily have one natural size; it does not degrade uniformly nor sometimes even monotonically.[6] In addition, it is not solely the amount of virtual memory traversed but the amount multiplied by the frequency that tends to cause paging explosions. And until it is known how much real space will be alloted to the program, it is difficult to decide how to give weight to the local amounts of virtual space traversed.

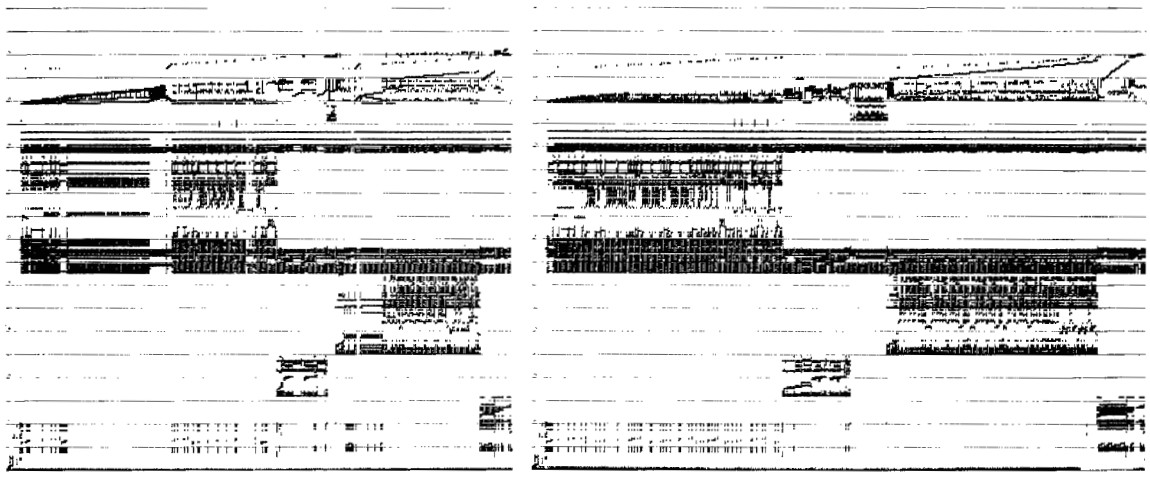
To some extent, the tendency of a sector to be found in physical memory when needed is related to the time since it was last needed. That time can be measured in terms of the number of intervening sectors needed (and should be measured in terms of the number of intervening pages). For instance, if the time is short since sector j was last referred to and little virtual memory space was used during that time, it is probable that sector j is still in real memory and a new reference will not cause a page exception. As the time between references to j increases, the probability of an exception increases unless the referencing

Figure 2 Memory usage during separate compilations



sector and j are in the same page. But here again it is difficult to assign a weighting function. It seems there is no need for the weighting function to be monotonic, since, if the time between references to j is very large, it is not worth placing j on the same page with the sectors that refer to it if this means displacing sectors referred to more often. But where should the weighting function peak, and what should be its value if the time between references is very small or is very large? At present, little is known about the desired shape of such a curve or the payoff for using a curve at all. Therefore, the thrust of the techniques described here will not be concerned with possible local weightings of the matrix during its generation.

The criticism that the nearness requirements among a set of program sectors is a very sensitive function of the data input to the program is more telling. There is no economy in tracing a program, massaging the data, reloading sectors, and measuring changes in paging rate if the improvements only hold for the particular set of data used when it was being traced. Fortunately, many commonly used programs are rather insensitive to data or respond in terms of overlays (new sectors) specific to specific variations in the data. For example, Figure 2 shows memory usage during FORTRAN compilations, with and without COMMON statements, dimensioned arrays, EQUIVALENCE statements, subroutines, calls to library mathematical subroutines, secondary storage, and console I/O routines. These plots give little hint of the functional differences among the source data compiled other than a stretchout of various phases of the compilation due to the number of source statements of a specific type. The horizontal axis represents execution time measured in units of 2500 instructions, while the vertical axis represents virtual memory at



TIME IN NUMBER OF INSTRUCTIONS EXECUTED

256-byte resolution ruled into 4096-byte pages. The vertical lines reflect the fact that the corresponding memory regions are in use. Similarly, the assemblers we have examined are not particularly sensitive to differences in instruction type. They call in new procedures and data areas to handle macro expansions, so that the macro phases of the assembler can be treated separately with respect to the generation of a nearness matrix and the ordering of extra procedure and data sectors. But it is certainly true, especially of application programs, that the uniformity of sector nearness requirements over a range of input data should be established before sector reordering on the basis of a nearness matrix is attempted.

reordering matrix C

Given the matrix **C,** possibly corrected for local variations from the time average, we wish to find a method of ordering the sectors, and thereby ordering the rows and columns of **C** so as to bring the largest c_{ij} values into square submatrices along the diagonal. These submatrices do not all have to be the same size, although all are square. Since the length of a relocatable sector can vary from a few bytes to a few pages, one would expect different numbers of sectors in the different virtual pages and therefore submatrices of sizes $s_1, s_2, s_3, \cdots, s_n$ with no restrictions on the integer values of the s_i. Of course, if only one sector or a fraction of a sector can fit within each virtual page, the effective virtual page size should be made some multiple of the actual virtual page size. One cannot cluster many one-foot cubes in a one-foot cubical box.

The assumption involved in clustering sectors into submatrices corresponding to one page (or one effective page) of virtual memory is not that there will be only one page space available

in physical memory. Instead, it is critical to minimize the links between any page out of physical memory and all the pages in it. In other words, once a page has been removed from physical memory, we wish to prolong as much as possible the time until it is needed again. Treating the problem at a one-page level corresponds to assuming that the working set of pages tends to change incrementally rather than in big jumps. If a working set of eight or ten pages were always to change completely within a few instructions but remain relatively undisturbed in between, we could reasonably cluster at the eight-page level instead of the one-page level.

matrix operations

We know of no efficient procedure to produce and prove the optimal ordering of the rows and columns of **C** to maximize the sum of the values in the diagonal submatrices,

$$\sum_{i,j\epsilon\alpha} c_{ij}$$

Several heuristic approaches give results that show only limited additional improvements when operated on by local perturbations. One method uses the eigenvectors of the matrix **C.** In each eigenvector, there are some elements that are (absolutely) large and others that are small. If the elements are taken in this order and a page is filled with the sectors they correspond to, one can associate a figure of merit with each eigenvector by comparing the components of the vector for sectors in the filled page with all the other components. Ideally, the vector will have large components associated with enough sectors to just fill up a page, and very small components elsewhere. In the case of a matrix that has all its nonzero values clustered in square submatrices about the diagonal, the eigenvectors will have nonzero values corresponding only to the members of those clusters. It is assumed that small variations in the values of the elements c_{ij} will produce smaller variations in the distribution of values in the eigenvectors. As the sectors associated with large values in an eigenvector are removed from the matrix, new eigenvectors can be calculated, and the process iterated until all sectors are assigned to pages.

An approach that gave slightly better results on the matrices that we examined can be visualized by considering the sectors to be physical weightless nodes. Assume also that the value c_{ij} in the nearness matrix is the strength of a spring connecting node i to node j.[7] For this analogy to be consistent, matrix **C** must be symmetric; we can insure symmetry by replacing c_{ij} with $c_{ij} + c_{ji}$ for all $i \neq j$. This does not vitiate the model since the requirement that i be near j is equivalent to the requirement that j be near i. If we fasten each node to the ground with a weak spring of strength g,[8] and lift the whole assembly by a node i, we are interested in the node i that pulls a few other nodes (just enough to

fit into a page with it) up close to it and leaves most of the other nodes near to the ground. For each node i that is pulled up to a height h_i, we need to know the heights of all the other nodes. This is a simple problem in statics, which can be solved by minimizing the energy of the system. It turns out that the relative heights of all nodes below node i are given in the ith row of the inverse of a matrix **D** constructed from **C** as follows:

$$d_{ij} = -c_{ij} \qquad \text{for } i \neq j$$

$$d_{ii} = \sum_{j=1}^{m} c_{ij} + g$$

The rows of **D** inverse are then rated using a figure of merit that compares for each row i the set of nodes that are raised close to node i and the remaining set. The size of the set of close nodes is determined by how many of the sectors will fit into a page with sector i. The figure of merit compares some function of the sectors in the page with that same function of the rest of the sectors. We have had equally good results with functions based on the values of **D** inverse and **C** itself. For instance, one can compare the heights d_{ij} of the sectors fitting into a page with the heights d_{ij} of all the excluded sectors, or compare $1/(d_{ii} - d_{ij})$ for $i \neq j$, or $d_{ij}/(d_{ii} - d_{ij})$. Using the nearness matrix **C,** one can compare the nearness values within the proposed page with the nearness values between those sectors in the page and all other sectors; specifically,

$$\sum_{i,\, j \epsilon \alpha} c_{ij} \qquad \text{with} \qquad \sum_{\substack{i \epsilon \alpha \\ j \notin \alpha}} c_{ij}$$

A cluster is made using the candidates in the best row, and the matrix $\mathbf{D}^{-1}$ is reduced by the rows and columns of the sectors in that cluster. Then the rows of **D** inverse are again ordered by the figure of merit, and the process of selection is iterated until all sectors are assigned. For a slight improvement, one can generate a new **D** from the reduced **C** and invert at each stage before selecting the best row of **D.** Ties are handled by tentatively making each assignment and comparing the next best figures of merit from the reduced versions of **D.**

There may come a point when for any possible cluster, the values of c_{ij} within the potential cluster are far less than the values from the cluster to those clusters already formed. At that point, the strategy changes to clustering so that the sectors all need to be near the same existing cluster, since clustering nearness across pages is better than no clustering at all.

holes in pages

When all sectors have been assigned, one problem remains: what to do about page boundaries. Holes in pages can come about because sectors do not fit evenly into pages. If sectors are not allowed to cross over page boundaries, there must be empty space within the page. The alternative is to pack the sectors

together across page boundaries, leaving no holes. Experience to date indicates the relative success of the latter approach. This is not surprising, since the presence of holes spreads the relocatable sectors over a greater virtual address space. This requires on the average more pages to be in physical memory at once for the same number of instructions executed without a page exception or, said differently, more page exceptions for the same number of pages in physical memory.

page boundary crossovers

If sectors can cross page boundaries, good choices must be made of what clusters are adjacent, since a sector that crosses a boundary will probably require that the clusters in both pages be in physical memory within a short time of one another. In order to take advantage of this confluence (or equivalently, to insure that both pages are in physical memory whenever the crossing sector is needed), we try to put next to one another the clusters that have the greatest nearness requirements. Since this involves merely sequencing and not clustering the clusters, the problem can be solved analytically.

If we think of the nearness requirements between any two clusters α' and β' as simply the sum of the c_{ij} from sectors in one cluster to sectors in another, or

$$\sum_{\substack{i\epsilon\alpha' \\ j\epsilon\beta'}} c_{ij} = a_{\alpha'\beta'}$$

we need only solve the maximal tour (traveling salesman) problem on the matrix **A**. This means that a circuit is made of the clusters so that the sum of the transition values $a_{\alpha'\beta'}$ between adjacent members of the circuit is maximized; corresponding to the circuit 1, 3, 2, 4, 5, 1 is the sum $a_{13} + a_{32} + a_{24} + a_{45} + a_{51}$. Since the last sector of code does not cross over to the first virtual page, the last term in the sum can always be set to zero. The choice of a first cluster is often not arbitrary, since the low addresses in virtual memory are generally used by control programs in a nonreloadable sense. Therefore, the circuit can be converted into a sequence without loss of rigor.

Optimizing solutions to the traveling salesman problem by the nearest city (in our case, farthest city) method have shown better paging performance than optimal solutions by the branch and bound method.[9] The reason for this may be that it is not equally important that all clusters be adjacent to their favorite neighbors but that this feature be biased in favor of the clusters most often in physical memory. These clusters each tend to have large values for

$$a_{\alpha'\alpha'} = \sum_{i,j\epsilon\alpha'} c_{ij}$$

and for $a_{\alpha'\beta'}$ to clusters β', which are also often in physical

memory. Thus initially biasing the traveling salesman problem by looking for the greatest valued $a_{\alpha'\beta'}$ in **A** can provide a favorable bias.

Code usage displays

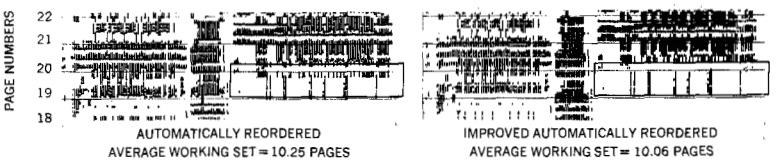
So far we have not taken into account the finer points of uneven usage of code within a module and the importance of choosing the best sector to cross a page boundary. Rather than automate the description of such complex situations, we have relied on computer displays to assist us in further reordering of sectors. Two principle advantages of on-line displays over the hard-copy output of a plotter are the potential for changing scale rapidly and the ability to see within seconds the changes in memory usage density resulting from sector reordering. Twenty to forty pages can be examined at low resolution and then those places where a frequently used sector crosses over into a page occupied otherwise with infrequently used sectors can be looked at in great detail. By associating the code usage data with individual sectors rather than with a particular ordering, the effects of a reordering can be displayed by simply reordering the retrieval sequence from the data file.

As was previously mentioned, there are cases when the nearness matrix alone does not have all the information needed for producing a good sector ordering. Fortunately in these cases the memory usage display is a help in deciding how to do relative scaling of the c_{ij}. The display gives an indication of the amount of memory space traversed within different periods of program execution; this space can be correlated with local paging behavior. Or it may happen that an infrequently used sector i is placed in the same page with a sector j that is used continually throughout the program. This is due to the fact that, although the sectors i and j are not often needed together, during the few times they are together, their activity is intense enough that the final value of c_{ij} is relatively high. In this case, the c_{ij} value is an inflated estimate of the need to have sectors j and i on the same page.

accounting for code usage

So far it has been assumed that the code within a sector is uniformly used, and that the sectors within a virtual page are uniformly used. This is often not the case. There are several things that can be done. If the low address or high address portion of a sector is used sparsely or not at all, that sector can cross the bottom or top boundary, respectively, of the page it lies in. The effect of lightly used code (such as some error-handling routines in compilers) is very like that of a hole of dead space. Both spread the heavily used code farther apart and can potentially cause extra paging.

Figure 3 Memory usage during third phase of compilation



It is often easiest to spot this condition in a display of memory usage over time. In Figure 3, the time quantum (associated with the x axis) is 2500 instructions, well within the time it took to bring in a page from secondary storage on the multiprogramming system used in this performance study. In this display, during the third phase of a compilation, virtual page 19 is occupied by two sectors. The more frequently used one requires that the page be in physical memory much of the time, but for most of that time, most of that page is effectively dead space. Placing the regularly used sector in a page with other regularly used sectors and the irregularly used sector in a page with other sectors used at the same times results in reduced page exceptions over a wide range of available real memory sizes.

In general, one must be careful to prevent frequently used code from extending even one byte over a page boundary, since the need for one byte can cause a page exception as surely as a transfer of 256 bytes into the middle of the page. And if there is an unused space in the middle of a sector surrounded by uniformly used code at both ends, the middle may as well look like the ends unless it is removed from the sector and combined with another.

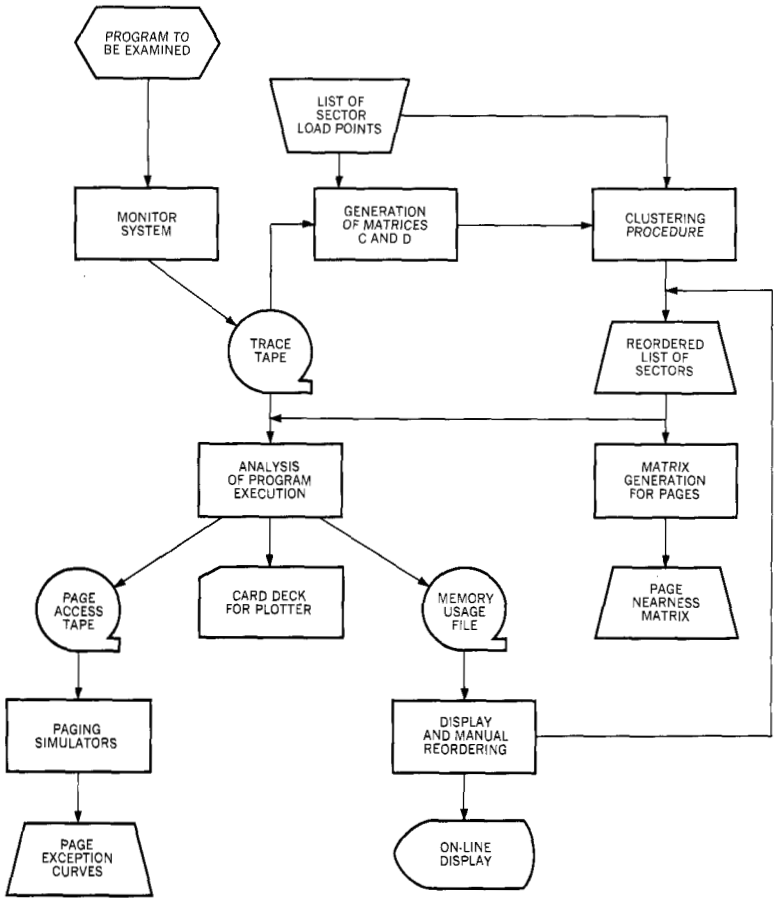
Displays permit the visual effects of real-time reordering to be examined immediately to insure that the effects of rearrangements over a few pages are nonnegative with respect to the other pages. For this phase of sector ordering, the use of computer graphics has produced an order of magnitude speed-up.

the reordering procedure

The techniques we have discussed so far may be considered as parts of an interactive process. The procedure followed is shown in the flow chart in Figure 4. There are two primary inputs to the process: the program to be reordered and the load map of the relocatable sectors that comprise it. This load map is usually the normal list of control sections given to a loader or linkage editor. In addition, it may include the starting point of procedures and data arrays that are normally transparent to the linkage editor.

First, a program trace is generated. Then two separate reductions are applied to the full instruction trace tape. Along one path, the nearness matrix **C** and then the derivative matrix **D** are produced.

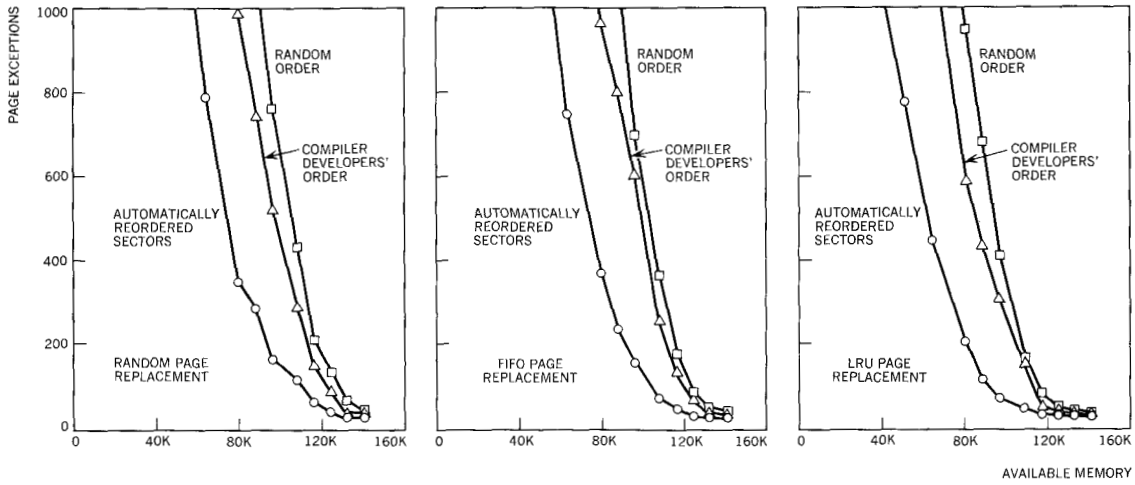
Figure 4 Sector reordering process



These two matrices, together with the sector load map, are input to the clustering algorithm, which finds an initial reordering of sectors for a given page size. This reordering can then be applied to the original nearness matrix to generate the nearness matrix for pages.

The other path begins with the application of a sector ordering to the trace data to produce some representation of program execution. A page access tape gives the (compressed) sequence of virtual pages of a specific size needed for program execution. This tape is then used as input to page replacement simulators, which measure the paging performance over a range of real memory sizes. A card deck can be generated for a plotter representation of the program's use of virtual memory over the course of its execution. For greater detail and additional sector reordering, this time by human decision directly, a data file is produced for on-line display, so that the detailed effects of uneven code density and page boundary crossover can be taken into account.

Figure 5 Paging performance for three sector orderings



Any sector ordering determined at this stage can be fed back into either path to produce a new page tape and a new page nearness matrix.

Paging performance

Generally, the paging reduction associated with the automatic sector ordering plus hand finishing have been in the range of two-to-one to ten-to-one.[10] We treat a specific example in some detail. The program involved is a highly modular compiler.[11] It has three phases and uses between 70 and 100 modules per phase, most of which are overlaid. It uses about 40 pages of virtual memory for procedures and data. The nucleus of the supporting operating system was also reordered.

Figure 5 shows the performance of three load orderings of the sectors of the compiler: a random order derived by randomizing the compiler order; the order used by the compiler developers themselves; and the automatic order, produced by clustering the nearness matrix for the compilation under examination. Three different page replacement strategies were used in each case: random, first-in first-out (FIFO), and least recently used.[12] The system nucleus was locked into storage for these tests. Figure 6 shows the page exception rate during program execution for the FIFO algorithm and the three different orderings. The automatic ordering was later improved with the aid of a memory usage display to give an ordering that produced about 20 percent fewer exceptions than the automatic over the range of real memory sizes. Figure 7 shows plots of memory usage for the random order, the compiler order, the automatic order, and the improved

Figure 6 Page exceptions for three sector orderings

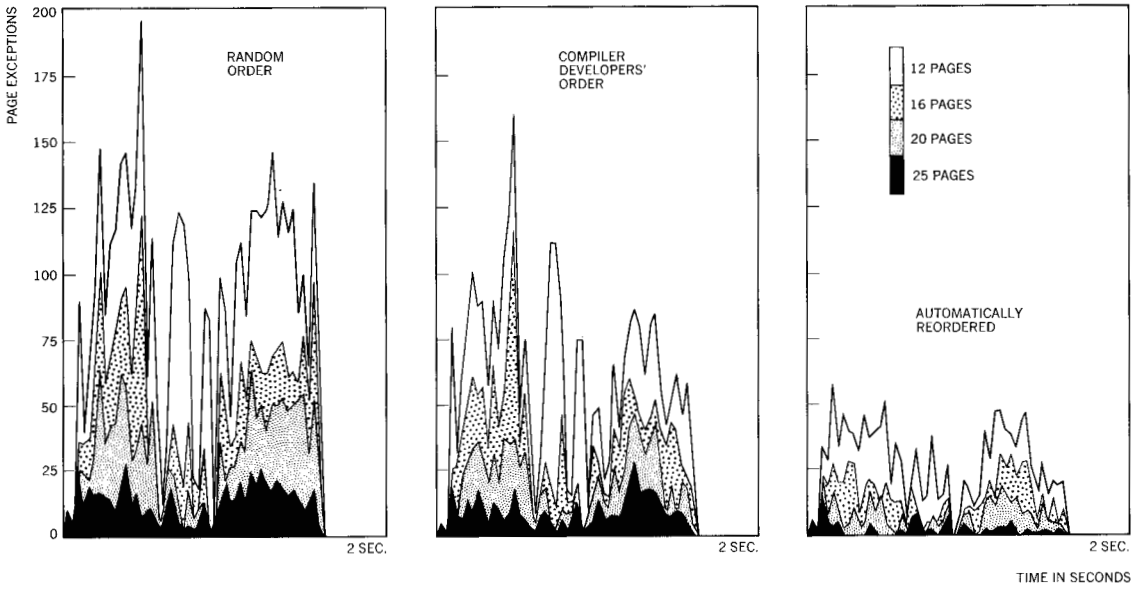
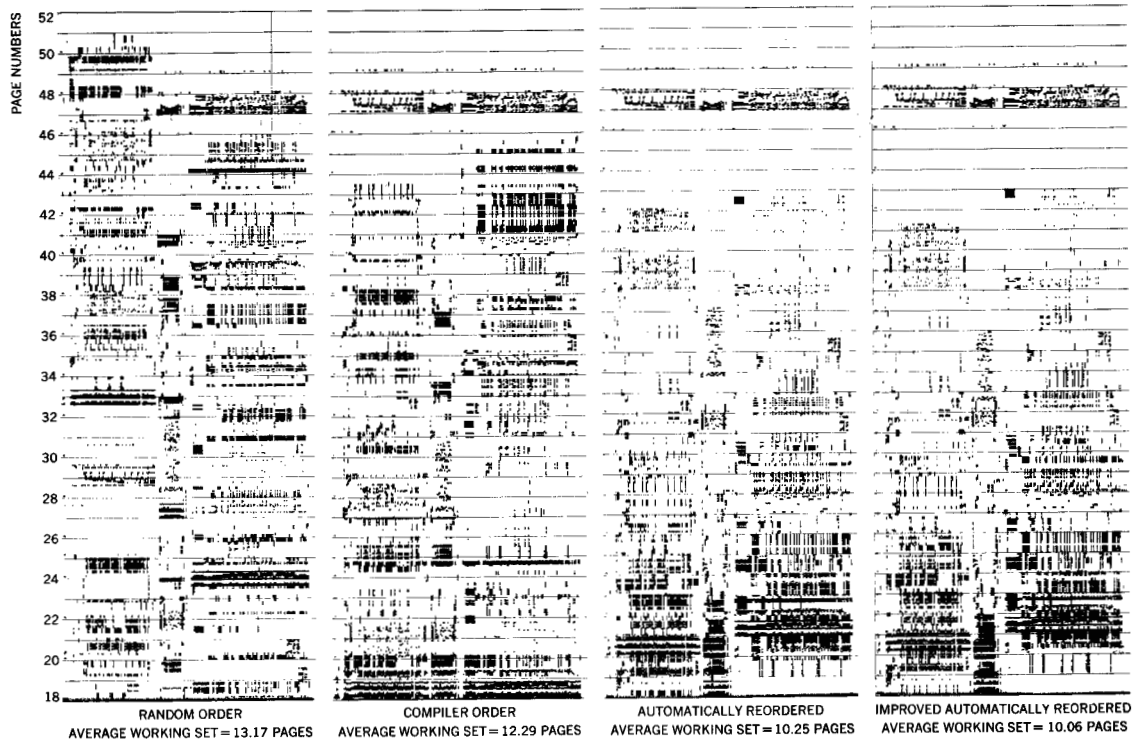


Figure 7 Memory usage for four sector orderings



automatic order, the principal evident difference between the last two being the overall use density within pages and the relation of memory use to page boundaries.

One characteristic of the relative performances of good and bad orderings (Figure 8) seems worth noting: the greatest improvement in performance occurs not at the severest real memory constraint but in the middle range. This can be (but is not necessarily) explained by the following situation: the better orderings not only concentrate appropriate sectors into pages, but these pages also naturally cluster into larger units that satisfy nearness requirements on the page level—and cluster better than do the pages of the other orderings. Therefore, when there is space for twenty or so virtual pages to be in physical memory together, both kinds of improvements are seen. But when there are only five or ten pages available, the effect of page clustering is much less important for all orderings and the effect of clustering sectors into pages alone predominates. This only means that the nearness matrix is such that clustering sectors into pages also clusters pages into larger units. Such a distribution of values in **C** is not improbable if the nearness requirements between each sector and all the others do not cut off abruptly at the number of sectors that will fill a page.

For a poor ordering, the nearness matrix for pages is much more uniform than for a good ordering; in the case of the good ordering with nearly enough real space, the real pages removed from physical memory are much less strongly tied to the pages in physical memory. But when there is very little real space, there are ties between pages in and pages out of physical memory for both orderings but for different reasons. The poor ordering often has sectors in one page that are most strongly tied to sectors in another page. The good ordering clusters sectors well, but now the well-clustered pages are split apart by the severe physical space constraint. The good ordering is still better than the poor, but relatively not so good as when there is more physical space. An examination of the page nearness matrices for the random and the automatic ordering gives support to this interpretation (Figure 9).

ordering evaluator

An (computationally) inexpensive evaluator of sector orderings is needed so that a new ordering can be estimated as better or worse than an old ordering without emulating paging performance over a range of physical memory sizes and page replacement algorithms. Given the requirement of independence of replacement algorithm, two data sources would seem appropriate: the nearness matrix for pages and the plot of memory use over time. The nearness matrix can potentially be used for establishing a figure of merit based on the probability $p_{\alpha\beta}$ of virtual pages being in physical memory together. Unfortunately, as previously indi-

Figure 8 Relative performance for different orderings and page sizes

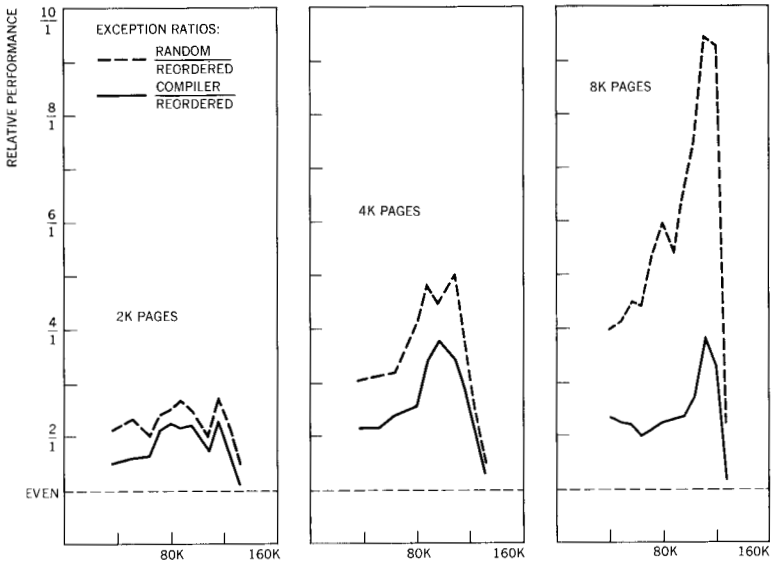
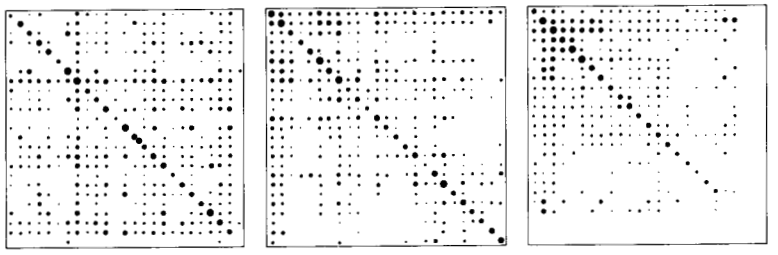


Figure 9 Page nearness matrices for three sector orderings



RANDOM ORDERING

PAGE SIZE	TIME QUANTUM	AVERAGE WORKING SET SIZE
4K	2500 inst	13.17
	5000	16.04
	7500	17.54
	10000	18.48
8K	2500	10.34
	5000	11.85
	7500	12.53
	10000	12.89
2K	2500	15.97
	5000	20.29
	7500	23.21
	10000	24.97

COMPILER ORDERING

PAGE SIZE	TIME QUANTUM	AVERAGE WORKING SET SIZE
4K	2500 inst	12.29
	5000	15.31
	7500	17.05
	10000	18.38
8K	2500	8.77
	5000	10.31
	7500	10.95
	10000	11.67
2K	2500	14.14
	5000	18.23
	7500	20.80
	10000	22.71

AUTOMATICALLY REORDERED

PAGE SIZE	TIME QUANTUM	AVERAGE WORKING SET SIZE
4K	2500 inst	10.06
	5000	12.73
	7500	14.21
	10000	15.32
8K	2500	7.57
	5000	9.19
	7500	10.02
	10000	10.59
2K	2500	12.72
	5000	16.33
	7500	18.79
	10000	20.57

SIZES OF DOTS REFLECT NUMBERS OF REFERENCES

cated, the rigorous solution for all pairs α and β in an m virtual page set in n physical pages involves the solution of $n\binom{m}{n}$ simultaneous equations, a solution computationally infeasible for the m and n we usually consider ($m > 20$ and $n > 5$). And so far, we have not found a reasonable simplifying assumption that will reduce the number of relevant initial states from $n\binom{m}{n}$.

Evaluations based either on only the diagonal of the page nearness matrix or only the diagonal and the $n - 1$ largest entries

in each row have failed to give figures of merit consistent with the relative performances under the paging simulator. Our experience with the memory use plots has been more encouraging. Since localization of virtual memory use implies the reduction of the size of the short-term working set of pages,[13] we can calculate an integral of the working set size as a figure of merit for a particular sector ordering, and get the average size by dividing that integral by the time for program execution.

Since the working set becomes exceedingly small for one instruction (averaging around two pages for the programs we have examined), the question of the magnitude of the instruction quanta arises. Again there are obvious reasons for choosing an interval of the same order of magnitude as the time to replace a single page. For the IBM System/360 Model 67 with an IBM 2301 paging drum, this figure is around 5000 instructions. Working with data at a time resolution of 2500 instructions and a memory space resolution of 32 bytes, the average working set size in pages was calculated for various sector orderings, and for time intervals of 2500, 5000, 7500, and 10,000 instructions. The evaluator is monotonic, in that figures of merit are lower for sector orderings that page less (Figure 7). The figures of merit are monotonic for page sizes other than the size for which the reorderings are performed (4K) although not as sensitive for the smaller page sizes.

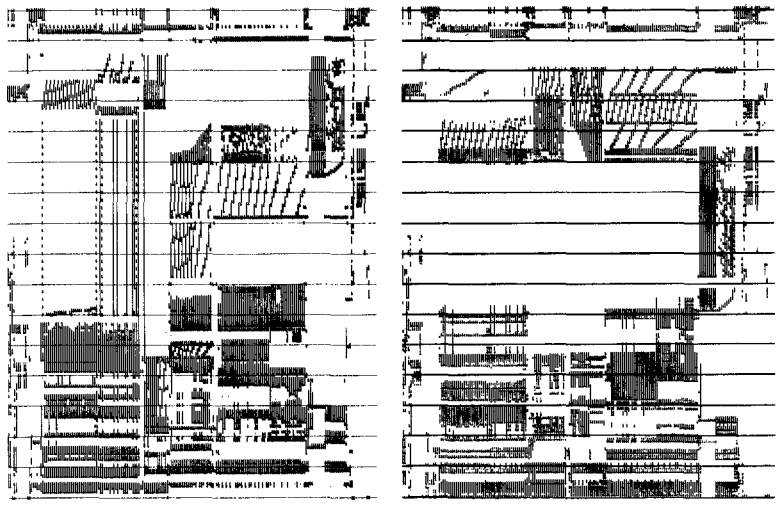
At present, work is continuing on this problem of calibrating estimators, including estimations of the joint probability $p_{\alpha\beta}$ that two pages will be in physical memory at once. In addition, the approach of packing sectors into pages so as to minimize the average size of the working set is being investigated.

defining new sector boundaries

One advantage of simulating performance on the basis of a trace rather than a real program is that low-density areas within sectors can be rearranged in virtual memory during the simulation without any recoding of the program. Since the simulator that generates the page request sequence for a given sector ordering does its own address mapping, sectors can be split and merged at will, as well as reordered. By defining new sector boundaries between the high-density and low-density areas of a sector, one can measure the effect of taking code out of line without recoding the program under examination. One can cluster these sectors, generated purely on the basis of density of code use, and get an estimate of the upper bound on the improvement possible without changing the programming strategy and/or storage strategy.

Displays of main storage usage are good indicators in themselves of the effectiveness of such strategies and are fast ways to spot the interaction of procedures and data once a hardware or soft-

Figure 10 Memory usage by two assemblers



LESS LOCALIZED MORE LOCALIZED

ware trace has been taken. For instance, the comparison of two assemblers processing the same source listing shows that while the techniques of one do not imply localized use of virtual memory, the techniques of the other while performing the same function generate more localized code (Figure 10).

Concluding remarks

Our general experience so far has shown the display of a virtual memory use pattern to be a good diagnostic tool. The automatic sector reordering technique brings noticeable improvements in paging performance where there is room for improvement, reducing the necessary working space (for a given number of exceptions) by as much as one-third to one-half. We have worked entirely in the environment of a simple hierarchy of main memory and a uniform speed replacement memory, with all replacement blocks having the same size. We have found that the reordering process, assuming a page size of 4K bytes, also produces improvements on pages of 8K and 2K bytes with the improvements favoring the doubled page size over the halved. This is consistent with a tendency we have noticed in the programs we have examined for better packed memory to favor larger page sizes. But the page size for clustering bears a direct relationship to the program sectors themselves. It has not proved effective to cluster at the physical page size when the average sector size is half as large or more. The optimal page size for a program depends on (besides physical I/O timings) complicated patterns in the use of virtual memory, about which very little is known.

The techniques that apply to a simple memory hierarchy also can be used for nested hierarchies with nested replacement sizes, such as paged memories with caches. Corresponding to the nested memories must be nested localization, with memory use first clustered at the smallest replacement size, then these clusters clustered at the next smallest replacement size, etc. Techniques must be tailored to particular hardware characteristics, such as a difference between the replacement size and the validation size of a memory, but the principles of compaction, even use density, and nearness in time requiring nearness in space still apply.

Not all problems arising from the over lavish use of memory can be solved by moving around sectors. Initialization of working space should be done incrementally as each page worth of space is used. Localized or "bucket" sorts are important. A sector that has different nearness requirements in different phases of a program's execution can be duplicated (which is easier to do if it is read-only) and put near the relevent code and/or data each time it is used. This increases the total virtual memory requirement but reduces the local virtual memory requirements. The principle that a program should go from one place to another with the fewest possible steps through distinct pages has obvious implications to list processing and other memory management strategies. In general, it is best not to anticipate the use of memory but to realize that where it is needed must be closely tied to when it is needed.

extensions

Besides the obvious extension to slightly different machine architectures mentioned above, it would seem appropriate to apply these packing techniques to the following areas.

Sector duplication, based on an analysis of the weighted communications matrix, may be desirable. Simply, if a read-only sector in one page can be duplicated in another so that the total communication within all pages is increased and between all pages is decreased without a major change in the number of pages needed "at once," such duplication should be performed. The relevant sector calls and returns must then be updated. Related to this is the possibility of duplicating specific sectors of the nucleus of an operating system so that the set of nucleus sectors needed for FORTRAN compilation would be optimally arranged with respect to one another. Similarly, the set needed for an assembly, a COBOL sort, an edit program, etc., would be rearranged. The compiler, assembler, sort, and edit program would each be using different copies of some parts of the nucleus.

A comparison of the number of communications among sectors to the number of communications within sectors can be used to measure the goodness of the modularity of the program.

Data areas in a program may be defined as sectors and their communications examined either through a trace or by examination of their nearness in the source listing. Thus, any statement referring to both data areas A and B is considered as a communication between A and B, and increments c_{AB} and c_{BA}. This intelligence could be built into an optimizing compiler. Such a compiler could be aided by its own or user-supplied estimates of the probability of taking any branch of code, and could insert such measurement probes into the compiled program.

Specifically, a slight modification of present compilers can produce information that will be valuable to the applications programmer in a relocation environment. The modification consists of the assembly language statements necessary to update a counter, or an element in a counter array, whenever a transfer is made to the code representing a labeled source statement. At the end of program execution, these counters could be dumped and the programmer could examine them in order to decide how to group the often used procedure sectors together. If, in addition, a variable is allocated to contain the number of the last labeled source statement active, an entire nearness matrix for the corresponding procedure sectors could be given to assist the programmer.

Considering the low cost at both compile and execution time and the value of the information, this modification would seem to make a useful optimization option for programs to be executed under relocation with dynamic address translation.

An intelligent compiler could be extended to reorganize its output object code and data into blocks for improved paging performance.[14] Decisions could be made about when to put data next to code and what data areas to put together. Eventually such a compiler should also address the whole problem of storage representation of data structures (such as pointer vs. matrix vs. hashed representation of a list data structure), since something is already known about performance in these areas. Assuming that software-implemented source languages remain popular for a few more years, there is an even greater need to provide a more sophisticated interface between the computer user and increasingly complicated software.

ACKNOWLEDGMENTS

The authors are grateful for the advice and assistance of the following members of the IBM Cambridge Scientific Center: R. J. Adair and P. V. Mockaptetris for adapting the tracing program to our particular needs; D. Ecklein, E. C. Hendricks, J. S. Moore, D. Tuttle, and W. Walker for support programs used in the generation of graphic displays; Y. Bard and S. G. Greenburg for discussions of the theoretical aspects of the problem; and

M. Dunn, M. Fleming, and other members of the operations group for generous support on an IBM System/360 Model 67 computer.

FOOTNOTES AND REFERENCES

1. P. J. Denning, "Virtual Memory," *Computing Surveys* **2,** No. 3, 153–189 (September 1970).
2. Some computers that have used or are using address translation for virtual memory facility or memory hierarchy for improved performance are the I.C.T. Atlas, the IBM System/360 Model 67, GE 645, IBM System/360 Models 85 and 195, X.D.S. Sigma 7, and RCA Spectra 70/46.
3. L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal* **5,** No. 2, 78–101 (1966).
4. B. S. Brawn, F. G. Gustavson, and E. S. Mankin, "Sorting in a paging environment," *Communications of the ACM* **13,** No. 8, 483–484 (August 1970).
5. L. W. Comeau, "A study of user program optimization in a paging system," ACM Symposium on Operating System Principles, Gatlingburg, Tennessee (October 1967).
6. L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characteristics of certain programs running in a paging machine," *Communications of the ACM* **12,** No. 6, 349–353 (1969).
7. H. R. Charney and D. L. Plato, "Efficient partitioning of components," SHARE/ACM/IEEE Design automation workshop, Washington, D.C. (July 1968).
8. A workable value for g in the context of the nearness matrices we have examined is $g \approx 2m$. In particular, see Charney and Plato, op cit.
9. J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel, "An algorithm for the traveling salesman problem" *Operations Research* **11,** 6 (November–December 1963).
10. Various runs of four compilers, four assemblers, two edit programs, and five applications programs have been examined. Execution time for the tracing program averages from thirty to sixty times the run time of the program traced, as does the time for generation of the nearness matrix C, or a page accessing tape. The clustering program takes from one to two minutes for a matrix of size 100×100. Since the work was done on a time-shared IBM System/360 Model 67, the tape I/O could be overlapped with CPU processing for other uses; run in batch mode, this I/O implies an additional 200- or 300-to-1 stretchout. For the example discussed in this paper, a total of four man-days of an analyst's time and one-half man-day of one of the compiler developers' time was involved. Five hours of computer time was used in tracing, generating matrices, reordering sectors, and displaying results. Page replacement costs on the System/360 Model 67 can result in a five- or ten-to-one stretchout per assembly or compilation.
11. The compiler is for AED, a computer aided design language developed under D. Ross at MIT. It is currently developed and maintained by the SOFTECH Corporation, Waltham, Massachusetts.
12. A total of eleven different replacement algorithms were used in comparing the performance of sector orderings, including all feasible (in the sense that they did not generate infinite loops) combinations of FIFO used and changed bits. In all cases, there was a marked reduction of page exceptions for the automatic ordering of sectors.
13. P. J. Denning, "The working set model for programming behavior," Communications of the ACM **11,** No. 5, 323–333 (1968).
14. Algorithms based on recognizing loops in source programs, keeping the loops from crossing page boundaries, and placing in sequence source program statements that transfer to one another have already been examined. See W. W. Ver Hoef, "Automatic Program Segmentation Based on Boolean Connectivity," *Proceedings SJCC* (1971).