Factors that affect the paging characteristics of user programs in virtual storage systems are presented in tutorial form. Measures are suggested that can be taken to exploit the virtual storage concept at the source language level in assembler, COBOL, FORTRAN, *and* PL/I.

User program performance in virtual storage systems

by J. E. Morrison

Much of the existing virtual storage literature has dealt with measuring, modeling, or otherwise anticipating system performance under paging with the goal of developing system control algorithms that can react intelligently in a wide range of situations.[1,2] The results of these investigations have laid the foundation for today's advanced virtual storage implementations. Some of these investigations may leave the impression that user programs are not easily controlled in the way they use real storage or, at least, that the programmer can contribute little to improving the system's overall paging characteristics. Historically this has been a reasonable assumption because until recently there have been relatively few virtual storage computing systems available. Therefore, few user programs have existed that were written with enough awareness of the virtual storage environment to truly exploit the technology.

This paper presents introductory principles and considerations for individual program performance in virtual storage and some programming techniques derived from them. The intent is to improve the performance of individual programs running in a virtual storage environment primarily, but not exclusively, through reducing their paging demands. The effect of paging on system performance and its dependence on the hardware and software configuration is not covered in this paper. In very large configurations, paging for a given program may be overlapped with the execution of other programs and therefore may not materially affect system performance. The entirely separate topics of system performance, configuration selection, virtual storage benefits that contribute to system throughput or programmer productivity, and new ways to design applications are not considered.

This discussion is motivated by several factors. First, regardless of the situation, the more a person is aware of his environment, the better he is prepared to take advantage of it. This does not mean that everyone who writes programs for virtual storage needs to be an expert on how it works, since one of the virtues of the virtual storage concept is that most of its benefits accrue without the user's awareness of its use. However, a program's operating environment can be profitably exploited given a minimal amount of background. An example of this type of exploitation is that of including facilities to take advantage of overlapped tape operations when multiple channels are available. The result is the same whether the user understands channel overlap and channel processing or not.

Secondly, many programmers write programs according to some set of standards imposed either by their organizations or on themselves by professional pride. Often these standards are completely arbitrary or are simply habits. For example, defining all FORTRAN arrays with COMMON statements is not an unusual practice among FORTRAN programmmers even though the same results come from the use of DIMENSION and using COMMON only for those variables that require it. Similarly, a common habit of COBOL programmers is to group all error processing routines together at the end of a program. Since many programmers use sets of standards, they might as well be standards that improve performance.

All material presented in this discussion pertains to the virtual storage concept as implemented by IBM, and is essentially free of specific operating system implementation considerations. It is assumed that the user write programs that reference a large *address space* divided by the system into fixed-length units called *pages. A page boundary* is any address evenly divisible by the page size. The computer on which the program runs has available to it a certain amount of *real storage* divided into page-size units called *page frames.* When the program is in execution, address-space references are mapped into real-storage addresses through the Dynamic Address Translation (DAT) facility. When a program reference is made to an address not resident in main storage, a *page exception* interrruption occurs signaling the need to load the referenced page from *external page storage.* The page frame that the referenced page will occupy is determined by the *page-replacement algorithm.* It is assumed that a page is never loaded until it is actually referenced (*demand paging*) and that only pages whose contents have been changed in some way are written to external page storage. It is further assumed that all multiprogrammed programs compete for the same main storage space so that higher priority programs are able to steal page frames from lower priority ones. The nature of virtual storage also dictates that the contents of those page frames being used

for Input-Output (I/O) operations must be fixed in storage for the duration of the I/O operation and are thus removed from the pool of page frames available for dynamic paging.

Principles of program performance.

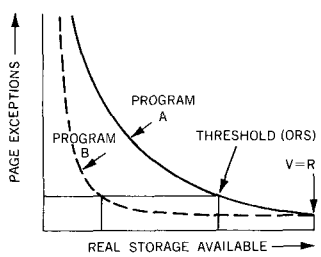
In the study of program performance in virtual storage system, two representations of the effect of paging have proved to be particularly descriptive:

- The parachor curve.
- The working set.

parachor curve

A parachor curve is a graph of the total number of page exceptions that a program causes to occur when paging against itself in a fixed amount of real storage, versus the amount of real storage available to it for execution. It is usually observed that if a program has less real memory available to it, then more page exceptions will result as shown in Figure 1. For most observed programs, there is a *threshold* point at which, if the amount of real storage is decreased even further, the number of page exceptions increases very rapidly.

Figure 1 Parachor curve



The parachor curve is descriptive in several ways. It shows how much real storage a program requires to maintain a "reasonable" level of performance (for example, a degradation of five percent). This area is shown as ORS (Optimal Real Storage) in Figure 1. The parachor curve also gives a subjective view of how sensitive a program is to real-storage limitations. In Figure 1, program A can be seen to be quite sensitive to the amount of real storage available over the entire range of storage sizes shown, whereas the number of page faults caused by program B is essentially independent of real storage until very near its threshold. Thus program B can be expected to be more predictable in multiprogramming situations.

The parachor curve for a given program is, however, a very gross measure of performance. Since the parachor curve represents a program paging against itself, its shape may be radically different in a multiprogramming situation where higher priority programs are stealing pages from it. Also, if a program that has occasional transient demands for many pages is multiprogrammed on a machine with very large main storage relative to its address-space requirements, paging rates will tend to be lower than shown in the parachor curve since the transient demand can be satisfied without requiring the program to page against itself. For some programs, the shape of the curve may be dependent on the data being processed although this is not nor-

mally the case. The most obvious weakness of the parachor curve is that it relates to the execution of an entire program, and the real memory requirements of a program change during the course of execution as does the amount of storage available to it.

working set

A more general description of how a program uses its address space is provided by the *working set*, the specific pages referenced by a program over some arbitrary interval of time. One can define a program's working set as $W(t, \Delta t)$, the pages referenced in the time interval $(t - \Delta t, t)$. The working-set size, $w(t, \Delta t)$, is the cardinality of $W(t, \Delta t)$.

Key to the definition of working set is the concept that a program does not have a single "working set" but rather has a series of working sets depending on the time at which the measurement starts and the interval over which it is taken. Obviously for $\Delta t = 0$, $W(t, 0)$ and $w(t, 0)$ are zero since no page references are possible in zero time. If one chooses to represent Δt in terms of instruction executions (which is admittedly nonhomogeneous in time, but easily measured on a computer), then as Δt increases one would expect $w(t, \Delta t)$ to grow in a fashion similar to the curve in Figure 2 where N is the total number of pages referenced during execution. N is not necessarily the size of the program and, in fact, is usually less since many programs never reference all of their address space. For example, some error routines may not be referenced in normal execution. Notice that no such general plot of $W(t, \Delta t)$ is possible for all programs because the specific page identifiers that constitute the working set are entirely program dependent; that is, if the program's total address space is 40 pages long but only 15 pages are referenced during execution, then it is irrelevant which 15 pages they are. The concern is the order in which those 15 pages are referenced.

Figure 2 Working set growth

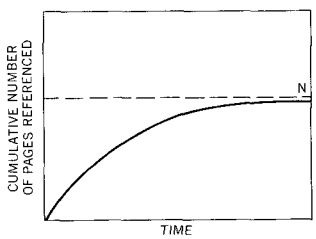
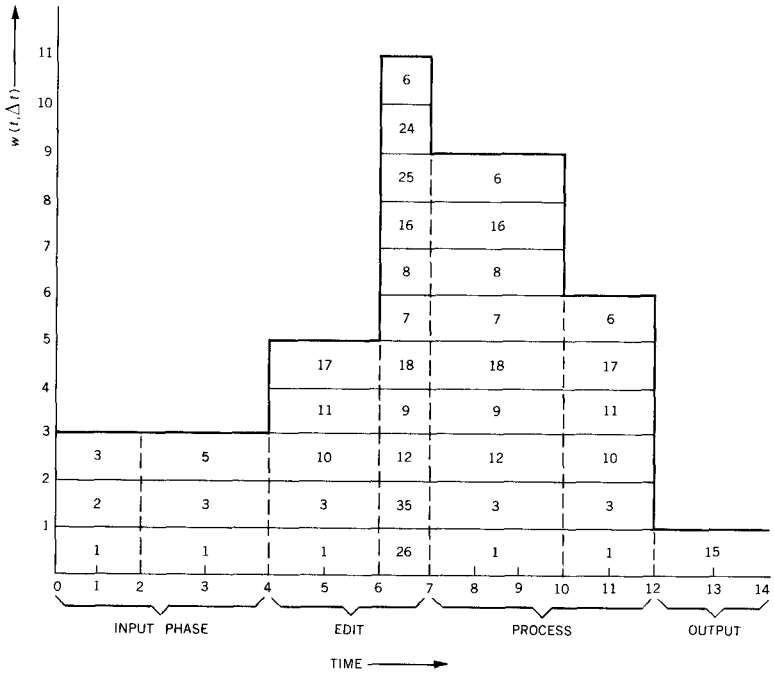


Figure 3 is a conceptual representation of the working sets of a hypothetical program. Assume a Δt interval of, for example, 1000 instruction executions. This figure illustrates the possible ways working sets can change during execution:

- The working set size may stay the same but the contents may change $[W(2, 2) = W(4, 2)]$.
- The working set may decrease in size $[W(10, 3) > W(12,2)]$ but paging will still result if the contents are not constant $[W(12, 2) \not\subset W(10, 3)]$.
- The working set size may change with no paging because the contents of the new working set were also members of the previous working set $[W(10, 3) \subset W(7, 1)]$.
- Both the working set size and contents may change $[W(6, 2)$ and $W(7, 1)]$.

Figure 3 Example working sets



These observations lead to the conclusion that one would like to keep the working-set contents as small and as constant as possible to minimize the demand for real storage and thus minimize paging.

locality of reference

A primary goal of programming for virtual storage is to achieve *locality of reference*. Although locality is difficult to define precisely, it can be subjectively defined as keeping a program's address-space references (both instruction location and data reference) confined to as few pages as possible for as long as possible. If good locality is achieved, then the working set will be both small and stable. Locality of reference is composed of three distinct parts:

- Internal fragmentation.
- Density of reference.
- Program reference patterns.

Locality is sometimes confusing because good locality of reference in no way implies that storage references must be close to each other; rather, the implication is that they concentrate in as few pages as possible. It makes no logical difference whether the working set is made up of one block of contiguous pages or an equal number of pages scattered throughout the entire address space.

Internal fragmentation, occurring in any storage management system where storage is allocated in fixed-size blocks, refers to the situation where a block is allocated but not completely used. An example is a 5K CSECT aligned on a page boundary, executing in a system using 4K pages. The result is 3K of unusuable real storage. Internal fragmentation creates a different problem than external fragmentation. External fragmentation occurs in systems where storage is allocated in variable length segments (as in OS/MVT) resulting in the problem of having enough total storage free to do something but not being able to use it because the individual fragmented areas are too small. External fragmentation results in storage that cannot be allocated. Internal fragmentation results in storage that is allocated but cannot be used.

fragmentation

There are two manifestations of internal fragmentation. *Physical fragmentation* occurs when program modules do not occupy an integral number of pages. In this case, the fragmented storage is never referenced. *Temporal fragmentation*, which has the same effect as physical fragmentation, occurs when the page is full of programs or data, and different parts of a page are not used together in time.

In order to differentiate between these two forms of fragmentation the temporal form of internal fragmentation will be referred to as *density of reference*. A page is densely referenced if most of its address space is used whenever the page is active in real main storage.

density of reference

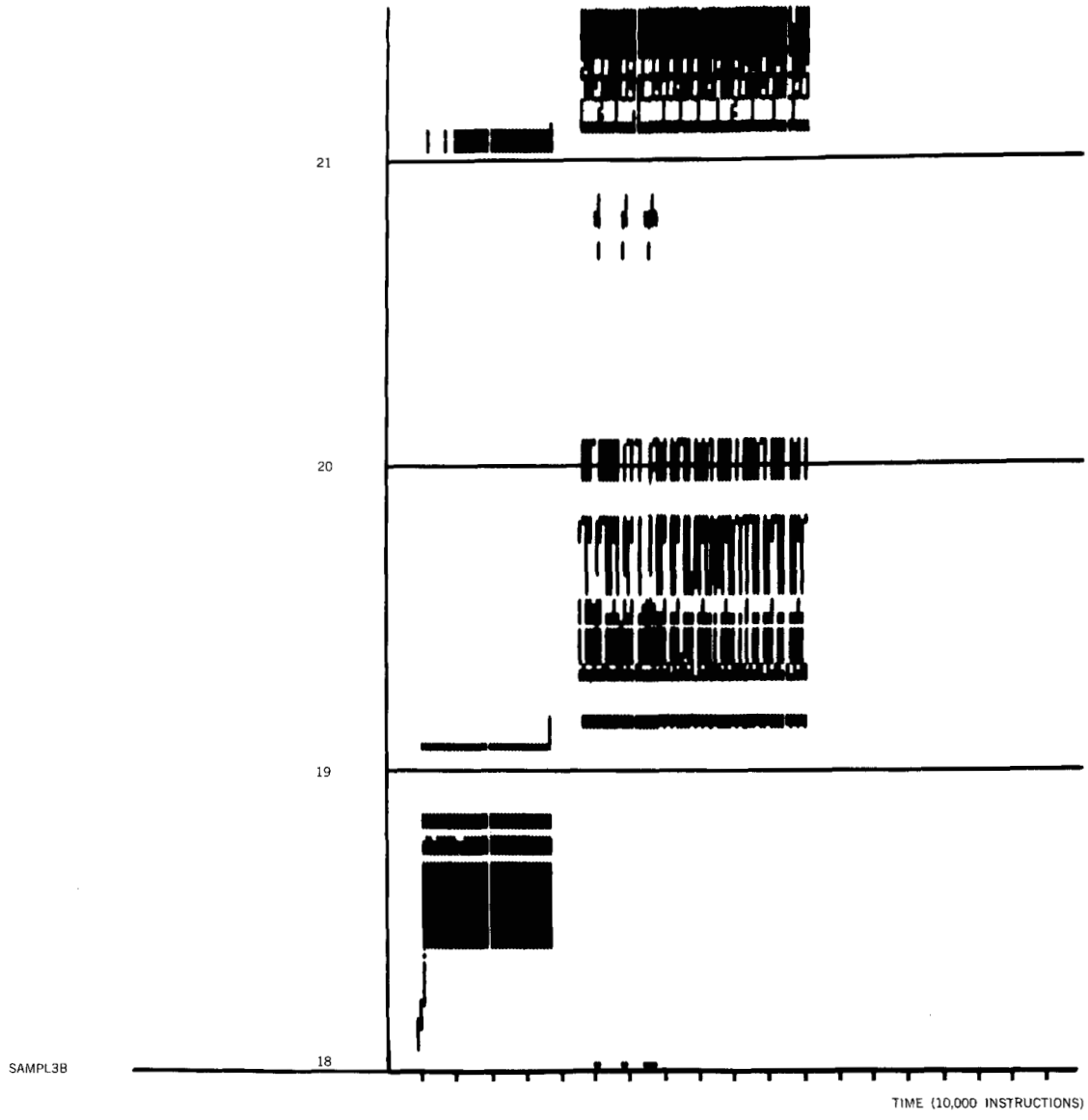
Another element of locality of reference is the manner in which address references are generated by the program. The *algorithmic aspect* of program design refers to the sequence in which data areas and executable code are referenced so that unproductive paging is avoided. In the context of this discussion, the placement of data areas is not an algorithmic consideration, but rather an element of fragmentation. Some of these techniques are discussed in subsequent sections.

algorithmic aspect

The concepts of locality, internal fragmentation, and density of reference can be illustrated by examples that show how a program can excel in one of the areas and still be a poor performer because it is deficient in the others. For these examples, and others that follow, a format developed by Hatfield that illustrates how a program uses its address space will be adopted.[3] These examples are sections of working programs.

In Figures 4 through 7 the horizontal scale is time measured in instructions executed, and the vertical scale is address space measured by bytes. Page boundaries are shown as solid lines and CSECT boundaries are shown in the left margin. The page size is

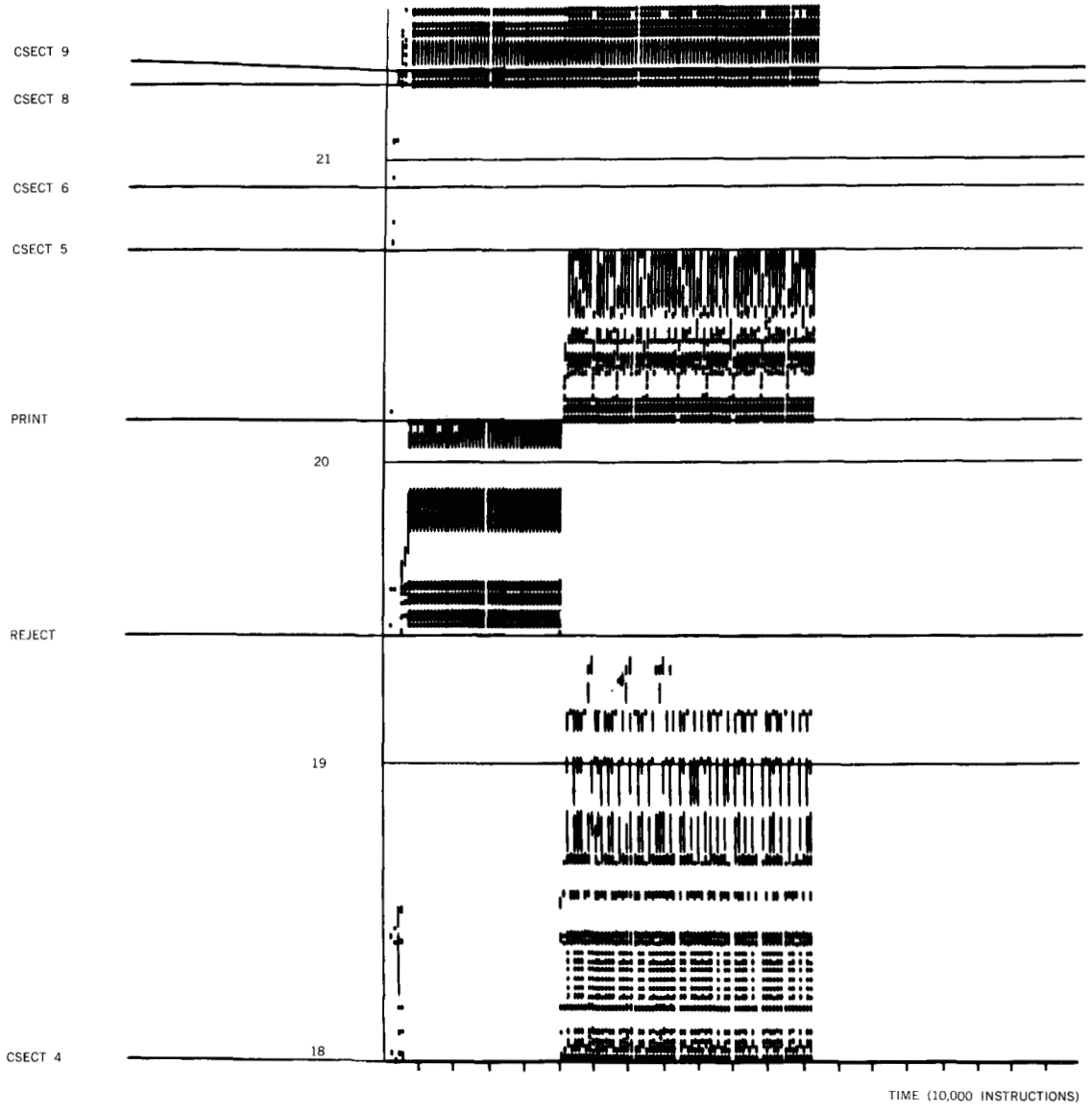
Figure 4 Fragmentation within a CSECT



assumed to be 4K. The dark areas are storage references. A solid dark area indicates concentrated references to a set of contiguous addresses. A dot or thin vertical line indicates a single or very small number of references to some location. It is assumed that, except for Figure 7, the time unit is the execution of 10,000 instructions.

Figure 4 shows how fragmentation and density of reference problems can occur within a single CSECT. This program section, which consists of executable code only, has two distinct phases, each with a different problem. The first phase is badly fragment-

Figure 5 Fragmentation between CSECTs



ed, using three pages rather than the single page actually needed. Page 20 in the second phase shows very poor density of reference.

The program in Figure 5 illustrates how fragmentation can result when CSECTs are combined. In this case, the CSECT labeled REJECT crosses a page boundary unnecessarily. By the simple expedient of exchanging the order of CSECTs REJECT and PRINT, a 4K page is saved through part of the execution.

Figure 6 depicts poor density of reference that can be corrected by packaging routines with concurrent execution into common pages.

Figure 6 Density of reference

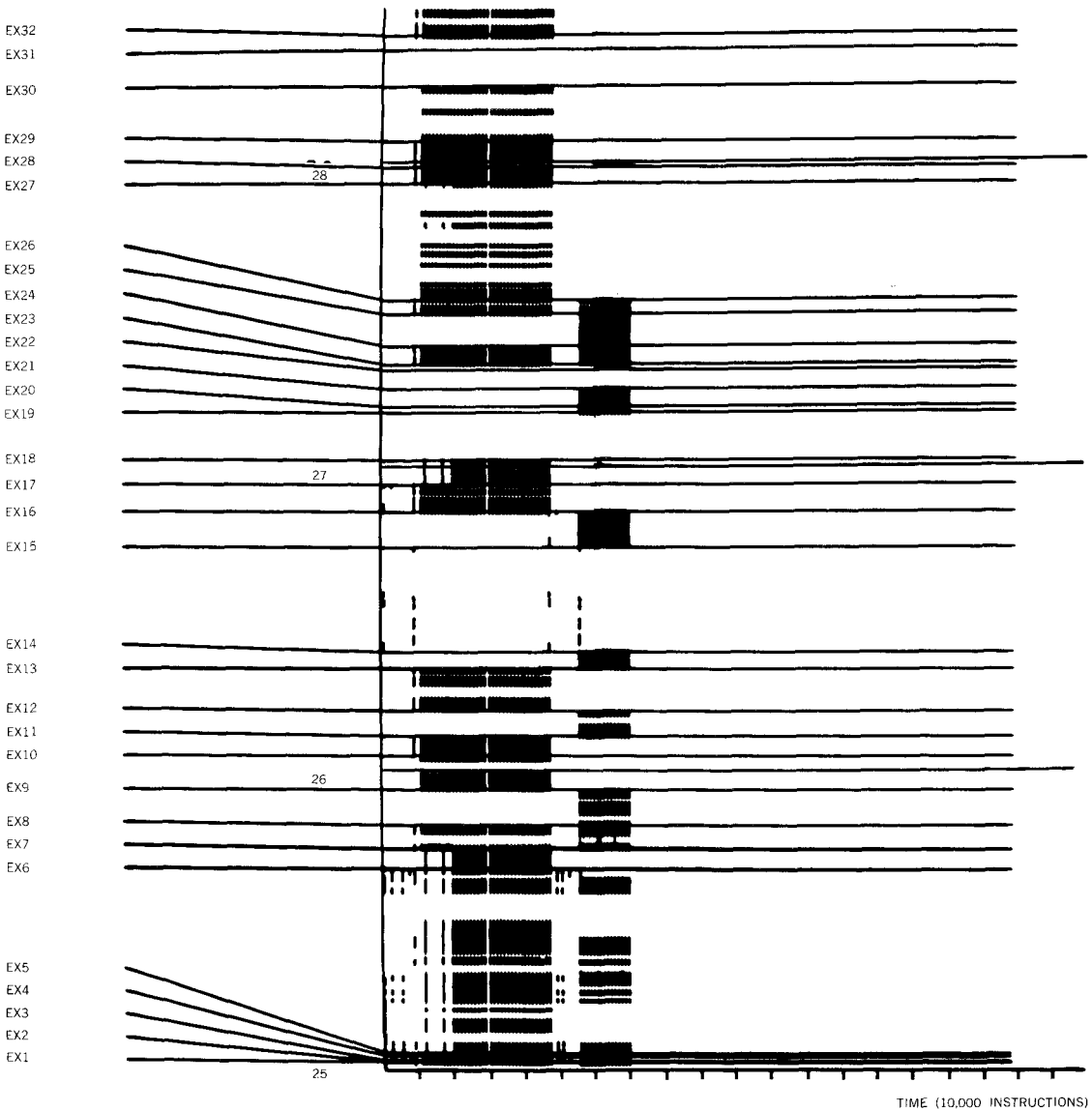
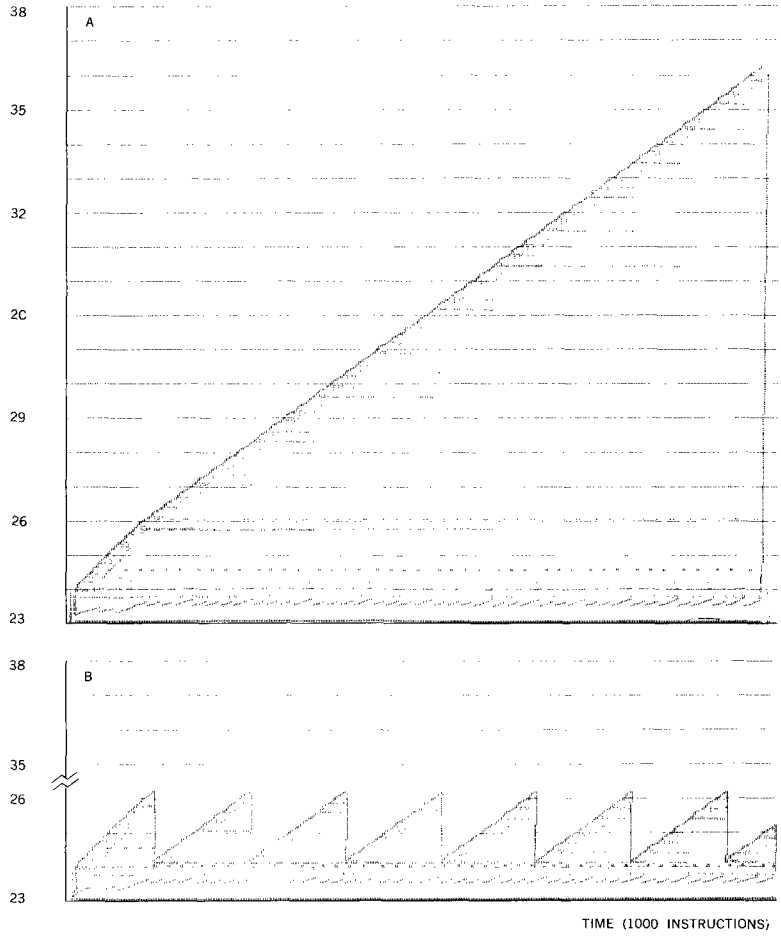


Figure 7 is an illustration of how a program's working set can be decreased by attention to algorithmic considerations. In this figure, the time unit is the execution of 1000 instructions. The process shown is the collection of free storage blocks during execution of an interactive program. In Figure 7A, the process covers several pages over a fairly short period of time, and covers the same pages over a very short period when the data is collected. The working set can be reduced significantly, as shown in Figure 7B, by simply performing the operation more frequently.

Figure 7 Algorithmic aspects



The objective, in summary, is to control a program's working-set dynamics in such a way that minimum paging results. This can be accomplished by concentrating on locality of reference, internal fragmentation, and density of reference. Techniques for achieving these results are discussed in later sections.

multiprogramming

In multiprogramming systems, programs compete with each other for system resources such as devices, CPU time, channel time, and (in OS) the SVC transient areas. In virtual storage systems, since the level of multiprogramming is likely to be high, another consideration arises—the availability of page frames to meet transient demands for storage from one or more concurrently executing programs.

The parachor curve, shown earlier, demonstrates how a program's paging performance is affected by varying the amount of real storage available to it. The implicit assumption in this curve

is that the program is running in a fixed amount of storage and is paging against itself. In a multiprogrammed system, the shape of the curve can be drastically changed depending on the real-storage demands of concurrently executing tasks with higher priority, and how well the page-replacement algorithm treats this particular job. The stability of the program's working set, while shown to some extent in the parachor curve, becomes more critical when multiprogramming because pages not referenced for short periods of time are more likely to be lost if the working set requirements of concurrently executing programs do not vary in a compatible way.

It becomes somewhat more crucial in virtual storage systems to not only balance the CPU and I/O workload, but also to match working sets. A study by Brawn and Gustavson at IBM'S Research Division indicates that optimizing programs for virtual storage to minimize real-storage contention during multiprogramming results in greater total throughput gains than are realized from the performance improvements of the individual programs.[4] In an operating system that deactivates low priority tasks in periods of heavy paging, the result should be fewer deactivations in systems with relatively small main storage capacities.

Program design for virtual storage

The preceding discussion presented some of the performance considerations and concepts implicit in the virtual storage concept. The subject of how to exploit these concepts is now addressed. The topic of reducing the paging demands of individual programs is emphasized because this is an area over which the application programmer can exert some control. This reduction is achieved by maintaining awareness that the program will execute in virtual storage and by using experience to develop program design guidelines and coding techniques that will exploit the technology. The payoff is a program that executes well in a broad range of main storage sizes and, hence, in a variety of multiprogramming situations. In addition, the resulting program may well be easier to design, code, and test because it is generally easier to apply virtual storage performance principles to programs after the fact than it is to develop a program to rigid storage constraints at any time.

There are two disciplines that can be applied to program development in order to exploit virtual storage:

- Packaging.
- Algorithmic techniques.

Packaging is the process of locating procedure and data areas in such a way that good locality, fragmentation and density of ref-

erence results. *Algorithmic techniques* can be used to control the order in which data areas are used or procedures executed. In the following discussion these two disciplines are freely intermixed, but it is generally obvious which type is being considered. The results that can be expected from the use of these guidelines are not quantified because of the extremely wide range of possible situations. Among the variables that affect the ultimate improvement possible are the size of main storage, system options chosen (such as resident functions), the external page storage device, level of multiprogramming, the degree to which these and other guidelines are used, and the nature of the application. It is not possible, therefore, to assign a value to each suggestion given or to validate it in a consistent way. Precedence, which validates that such measures do have genuine benefit, does exist in the literature.[1,3-6]

principles of program design

The following can be considered a philosophy of program design for virtual storage systems. These general principles are essentially independent of program function or language.

1. Concentrate on programs that consume significant system resources. Program design techniques for virtual storage tend to be valid for all sizes of main storage, but offer greater potential in systems with high storage utilization. Programs that do not run often or long, or do not require a significant portion of available main storage, are clearly of less concern than programs that dominate the system or are scheduled to a deadline.
2. Emphasize the reduction of short-term storage requirements even if the program's total size increases. The objective is to reduce and stabilize the working set of each significant phase of the program. The program's total size is of less consequence since only the active phase uses real storage.
3. Do not attempt to capitalize on unique characteristics of individual operating system implementations. Programs of this type are generally more difficult to test and maintain, and thus defeat much of the concept of virtual storage. It is interesting to note, however, that many virtual storage implementations tend to favor programs written with good locality without any other measures being taken. Techniques such as "fixing through false reference" enhance performance artificially and at the expense of system throughout.[7]
4. Optimize the main line for the normal case.
5. Because locality of reference applies only to the working set, the important consideration is to keep storage references confined to pages that would normally tend to be in main storage at the same time. It is logically immaterial whether the working set is made up of contiguous blocks of

pages or of pages scattered throughout the address space. Experience has shown, however, that internal fragmentation is reduced if the working set is made up of contiguous blocks of pages.[8]

6. Use optimizing compilers wherever possible since they generate less code and therefore create less DAT overhead.

Programming for virtual storage

General guidelines for program design, regardless of the language in use, are now discussed. Each guidline presented is implementable using an assembler language and, to a somewhat lesser extent, high-level languages. Specific considerations for PL/I, FORTRAN and COBOL are presented in subsequent sections. The following list is not comprehensive as it is probable that more guidelines will be recognized as experience is gained with virtual storage systems:

1. Remove exception and error-handling routines from the mainline of the program to increase the density of reference of the most heavily used pages. Put these routines into pages of their own, if possible, because they will probably never be used in normal execution of the program. All error conditions should be examined in the mainline to avoid unnecessary entry into another page. Low-use code that is not exception code (such as housekeeping and initialization routines) should be inline unless the routines are so big that the density of reference is materially affected.

2. Initialize each data area just prior to its first use rather than initializing all at the beginning of the program. This may prevent pages containing data areas from being loaded unproductively. In addition, if a large area is reserved so that a worst-case condition can be handled, it should not be initialized until it is known how much is needed.

3. Reference data in the order in which it is stored and/or store data in the order in which it is referenced. This is particularly true of arrays. If an array is stored by columns (as in FORTRAN), complete all references to a single column before moving to the next. The order in which data areas are referenced is, of course, of no consequence if the entire area fits into a single page. Most page-replacement algorithms tend to favor pages that have been used recently. Therefore if a procedure causes a large sequential span of storage to be traversed, the direction of scan should be reversed in alternate passes.

4. Store data as close as possible to other data used concurrently, and in the same page if possible. Also, store data used only by specific subroutines along with the subroutine

code, especially if its size is significant with respect to the size of the page.

5. Group high-use buffers and data areas together in common storage. This can be accomplished in assembler language by the use of COM and Q-type address constants.

6. Align large buffer areas to page boundaries. Since buffer areas are fixed in storage during I/O, careless alignment can result in extra pages being fixed. The most efficient size for buffers, from a paging standpoint, is the length of a page aligned on a page boundary. Other factors such as file packing, I/O transmission time, error recovery, and the possibility that very large I/O buffers may cause extra paging should be taken into consideration when selecting buffer size. Note that using fewer large buffers reduces the amount of channel program translation.

7. If possible, separate read-only data from areas that will be changed. This could save page-out operations in a highly utilized system since unchanged pages are never written to external page storage. Notice that this conflicts with Guideline 4. Generally locality considerations are more important to consider than strict separation of read-only and read-write areas.

8. When using assembler language, avoid the use of literals and literal address constants unless precautions are taken to insure that the literals are inserted in the same page. This can be accomplished using LTORG (literal origin).

9. Put seldom-used subroutines inline if they are not so big that poor density of reference results. An alternative is to put all the low-use, but required, subroutines into a page of their own.

10. Put subroutines that have nested calls in sequence. That is, if main calls A, calls B, calls C, then put A, B, and C together.

11. Avoid the use of elaborate search strategies for large data areas. Avoid the use of large, linked lists if these techniques cause a wide range of addresses to be referenced. Methods of using list structures are referenced.[9] The use of binary search for sequential tables spanning many pages should be carefully evaluated. Useful alternatives to binary search are hashing entries for direct access, or resequencing the table by frequency of use so that a sequential search may be used.

12. Send subroutine arguments by value, if possible, instead of by address.

13. If a subroutine repeatedly references storage in a different pattern than the rest of the program, compact the data to a workable format before the subroutine starts instead of gathering the data during the process.

14. Consider segmenting large arrays and data areas into page-size units and processing the segments instead of the whole area. For a detailed discussion of this topic, refer to McKellar and Coffman.[10]
15. Consider using virtual storage for scratch files to reduce I/O overhead and page fixing. This technique must be evaluated in the context of the program, the size of the file and its frequency of use.
16. Segment programs that have well-defined, long-running phases by function even if it means duplicating code to improve working-set stability.
17. Make common data areas more productive by using the same area for different data in the different phases of a program. This is of particular value when several independent subroutines using a common data area can share the storage for their unique work areas.
18. Consider batching input to a computational subroutine (for example square root or trigonometric functions) by "looking ahead" to see what values are needed and computing the results all at once rather than making separate calls for each argument to avoid repeated calls.
19. Whenever possible, use the subpooling technique for assigning dynamic storage. When properly implemented, this technique insures that items in the same subpool are stored close together.
20. If possible, avoid situations where instructions or operands cross page boundaries as this can cause extra address-translation overhead and the loading of extra pages.

High-level languages

The program design guidelines presented in the previous section are generally applicable regardless of the specific programming language used. Each language does, however, present a different program structure that should be considered when designing programs. The particular areas of interest are modules created and included by the compiler, and the method of data storage employed. Knowledge of how high-level language programs are structured and how data areas are located enables control to be exerted by the programmer to improve locality of reference. For example, COBOL data areas that have concurrent use can be located close together by defining them sequentially. The following high-level language guidelines are intended to accomplish the same purposes as those presented previously. Three widely used languages are discussed: FORTRAN, COBOL, and PL/I. Since there is some dependency on the specific compiler implementation,

the compilers considered are OS FORTRAN IV H Extended, ANS COBOL Version 4, and the OS PL/I Checkout and Optimizing Compiler.[11,12,13]

FORTRAN

There are several devices in the FORTRAN language that can be used to improve virtual storage performance. Two aspects of FORTRAN are particularly pertinent:

- FORTRAN stores arrays by column.
- The data areas that are reserved in the module produced by the FORTRAN compiler are in different main storage areas for COMMON, working storage (DIMENSIONed arrays and self-defining variables), and FORMAT statements.

Consider the following fragment of a FORTRAN program that initializes an array to zeros.

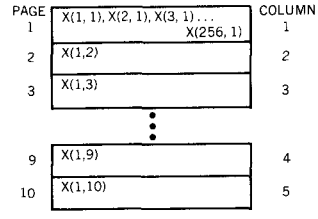
```
   REAL*8 X (256,10)
   DO 10 I = 1, 256
   DO 10 J = 1, 10
10 X(I, J) = 0
```

Each column of the array X will take up exactly one 2K page ($8 \times 256 = 2048$). Notice that, although the array is stored by columns, it is being zeroed by row, which means that a different page is being referenced for every single execution of line 10 (see Figure 8). By the simple expedient of changing the DO loop to work in the array by columns, it can be coded as:

```
   DO 10 J = 1, 10
   DO 10 I = 1, 256
10 X(I, J) = 0
```

Thus the operation of the loop concentrates in one page at a time instead of ten.

Figure 8 Array Storage in FORTRAN



It is more difficult to keep array references in storage order for more complex operations. Matrix multiplication requires one of the arrays being multiplied to be referenced in the wrong way as shown by the following program fragment that calculated the $M \times N$ inner product, C, of matrices A ($M \times L$) and B ($L \times N$). (Note that matrix C is filled in by rows):

```
   DO 20 I = 1, M
   DO 20 J = 1, N
   SUM = 0
   DO 10 K = 1, L
10 SUM = SUM + A (I, K)*B(K, J)
20 C(I, J) = SUM
```

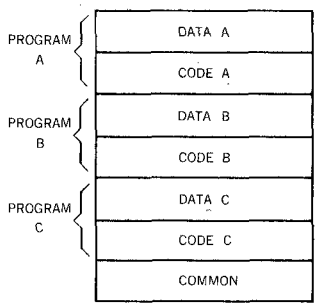
This routine can be improved by causing references to columns of B and C to proceed such that all references to one column are made before moving to the next:

```
      M1 = 0
      I3 = -1
      DO 20 J = 1, N
      DO 20 I = 1, M
      SUM = 0
      I3 = -I3
      M1 = M1 + I3
      DO 10 K = 1, L
      SUM = SUM + A(I, M1)*B(M1, J)
10    M1 = M1 + I3
20    C(I, J) = SUM
```

In addition, this code reverses the direction of scan across rows of A on every pass. The effectiveness of reversing the direction of scan (and thus the penalty for including the extra code) depends on the size of the array and the priority of the program.

The paging efficiency of many algorithms can be improved by considering the amount of parallelism possible in the computation and then using a storage reference pattern that exploits the parallelism to minimize working-set requirements.[6,10,14]

Figure 9 Representation of FORTRAN object module



FORTRAN creates separate CSECTs for each separately compilable module. Each procedure CSECT contains code and local variables. Other CSECTs in a FORTRAN program are named COMMON, blank COMMON and library subroutines.

FORTRAN's use of different areas for local variables and COMMON allows programs to be segmented by function. Working storage is a part of the module generated by the compiler and stays relatively close to the generated code. COMMON appears at link-edit time and is always the last (highest address) area in the load module as depicted in Figure 9. This organization allows us to pool data used by all programs in COMMON and segregate data used in specific routines in a very natural way.

One can take advantage of this by grouping data into the data area closest to the code that uses it, instead of arbitrarily putting all data in COMMON. Additionally, if the problem logic permits, one can segment a large mainline program into several smaller programs, each with its own working storage area as a method of locating data near the code that uses it.

When designing FORTRAN programs, one should be aware that although compiler-generated code is generally read-only, the page containing the code may be changed because implicitly-defined variables, loop-control work areas, address constants, and other internal variables are compiled into the same module as the code. This organization restricts efforts to separate read-only and changed areas for small routines. Other guidelines for use in FORTRAN programs are:

1. Define explicit variables in an order such that data items used together are defined sequentially.
2. Consider separating high-order dimensional arrays into arrays that have only a single plane so that they can be better stored for the sequence of accesses across them.[10]
3. Compact sparse arrays to conserve real storage and reduce fragmentation.
4. EQUIVALENCE all possible areas to save real storage in COMMON.
5. Constants and literals are stored in a unique area. Therefore, assign constant values to local variables at the beginning of the program to achieve better locality of reference.
6. Because FORMAT statements are stored separately, consider putting all formatted READs and WRITEs in a separate subprogram and then grouping I/O action. Avoid the use of implied DO loops in I/O statements as this causes repeated returns to the calling program.
7. Use statement functions instead of external subroutine calls for short functions as these functions are compiled directly into the module.
8. Group code so that it uses data from areas that are close together.
9. Reduce calls to library subroutines by using recursive definitions such as X*X instead of X**2.[14]

COBOL

Of the widely used, high-level languages, COBOL at first appears to offer the application programmer the least flexibility for virtual storage tuning. The very nature of the COBOL language encourages the programmer to write rather large, monolithic programs with well-defined mainline logic which is exactly what one would like in a virtual storage system. It would appear that application design techniques that capitalize on virtual storage concepts (for example, the use of virtual storage for work files and large tables) would be the most fruitful area to investigate. On the other hand, virtual storage can improve the performance (with respect to storage utilization) of many COBOL programs because many COBOL programs have error routines that are seldom executed.

Some specific coding techniques follow:

1. Group FDs by function and define files that are used together in sequence.
2. Files that are used together should be OPENed in the same statement. This will cause their buffers to be close together and improve locality of reference as buffers are processed.

3. Do not use ALTERNATE AREAS unless needed for a special reason. The net effect of the alternate buffer is to separate data areas.
4. Avoid the use of SAME RECORD AREA, as this causes buffers to be separated from files not using this option.
5. Group references to DIRECT files because buffers for these files are stored separately from sequential files.
6. Working storage is allocated in the order in which areas are defined. Thus organize working storage according to the rules already given, such as putting items that are used together in nearby definitions.
7. Exercise good COBOL coding techniques in USAGE clauses and length definitions to avoid the overhead and potentially poor density of reference caused by unnecessary execution of conversion subroutines.[12]
8. Put all exception-handling routines together at the end of the program or, if possible, in a separate program. Exception-handling routines should be ordered in the frequency of execution.
9. If virtual storage is used to store a work file, define that area either first or last in working storage to avoid separating other working storage items from each other by a large span of addresses.
10. In ANS COBOL, the segmentation feature can be used to align PROCEDURE DIVISION code. Using this feature, a separate CSECT is generated that can then be aligned to a page boundary or sequenced by the linkage editor to minimize fragmentation. Since all buffers and data areas stay in the root segment, it may be desirable to break up WORKING STORAGE by coding the called program as a subroutine.
11. The COBOL facility for dynamic loading and deleting of modules should be evaluated carefully before use, as this is accomplished by the virtual storage concept.

PL/I

PL/I offers the programmer function and flexibility in program design, and also offers many features that can be of value when considering virtual storage.

PL/I generates CSECTs under the following circumstances:

- For each PROCEDURE (Two CSECTs are generated—one containing only code and another containing INTERNAL variables).
- For each variable declared as STATIC EXTERNAL.
- For each file declaration.

- For compiler-generated subroutines.
- For library subroutines.

These CSECTs can be sequenced by the linkage editor to improve locality of reference and to reduce internal fragmentation. Additional pertinent facts concerning PL/I that are of value when designing programs are:

- PL/I stores arrays by row.
- Some PL/I functions, such as record I/O, cause subroutines to be loaded dynamically.
- Automatic storage, which is the default, is formatted every time a different procedure is entered.
- All PL/I code, whether declared reentrant or not, is read-only.

The following are guidelines for use in PL/I programs designed for virtual storage systems:

1. Define variables used by specific subroutines as STATIC INTERNAL so they will be close to the code that references them.
2. Define every variable possible as STATIC. This is particularly true for arrays and data structures.
3. Avoid the use of the INITIAL attribute in automatic storage.
4. CONTROLLED and BASED storage offer considerable flexibility in structuring storage. Locality can be controlled by judiciously assigning areas to BASED variables chosen so that areas that have concurrent use are assigned to the same variable.
5. Multitasking requires extra real storage.
6. Make variable declarations consistent across and within procedures to avoid unnecessary execution of service subroutines.

Organizing existing programs

Since program performance problems in virtual storage are often the result of abusing the virtual storage concept by undisciplined programming practices, existing program libraries do not often require extensive modification. If, however, it becomes necessary to consider a particular situation, the program organization principles stated earlier can be applied to programs that were not originally designed for virtual storage. In many cases good results can be achieved by determining which are the high and low-use modules, defining page-alignment requirements, and link editing

the program modules in the order that gives the best locality of reference. In this procedure, no code changes are required. High and low-use modules can be determined with program traces, entry counters, or from knowledge of the dynamics of the program.[3]

If the program had been in an overlay structure, and the overlays removed, the segments should be aligned to page boundaries. However, it is possible for an overlay program that has good locality in its segments, very dense use of overlay pages, long execution times in segments, and changes in most of the overlay pages, to have a performance advantage over the same program with its overlays removed. The exact advantage depends on the operating system implementation, the program logic, the overlay segment length, and the system configuration.

When packaging an existing program for virtual storage, it may be necessary to break some modules into separate modules or to move code around within a module to increase density. It is seldom necessary to otherwise recode routines unless program techniques are used that cause wide ranges of address space to be referenced.

Compiler-dependent service subroutines can also be packaged by function. For example, I/O-related routines can be put together as can related data conversion routines. Packaging order can be determined through the use of program traces, cross-reference listings, and Program Logic Manuals. Most service routines are fairly small; hence several can profitably be combined into a page.

Concluding remarks

Introductory principles of program performance in virtual storage have been presented to give a perspective of the virtual environment for systems designers and programmers. It is recognized that the list of guidelines presented is not comprehensive as many more useful techniques will be developed as experience is gained with virtual storage systems. It is hoped that these program design considerations and the programming guidelines will assist these users in their future virtual storage program development.

ACKNOWLEDGMENT

The author wishes to acknowledge D. J. Hatfield whose guidance and helpful comments were invaluable during the writing of this paper.

CITED REFERENCES

1. L. A. Belady, "A study of replacement algorithms for a virtual storage computer," *IBM Systems Journal* **5,** No. 2, 78–101 (1966).
2. An extensive annotated bibliography of virtual storage literature is contained in R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual storage and virtual machine concepts," *IBM Systems Journal* **11,** No. 2, 118–130 (1972).
3. D. J. Hatfield and J. Gerald, "Program restructuring for virtual memory," *IBM Systems Journal* **10,** No. 3, 168–192 (1971).
4. B. S. Brawn and F. G. Gustavson, "Program behavior in a paging environment," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **33,** 1019–1032 (1968).
5. B. S. Brawn, F. G. Gustavson, and E. S. Mankin, "Sorting in a Paging Environment," *Communications of the ACM* **13,** No. 8, 438–494 (August 1970).
6. A. A. Dubrulle, "Solution of the Complete Symmetric Eigenproblem in a Virtual Memory Environment," *IBM Journal of Research and Development* **16,** No. 6, 612–616 (November 1972).
7. R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual storage and virtual machine concepts," *IBM Systems Journal* **11,** No. 2, 99–130 (1972).
8. D. J. Hatfield, "Experiments on Page Size, Program Access Patterns and Virtual Memory Performance," *IBM Journal of Research and Development* **16,** No. 1, 58–66 (January 1972).
9. R. R. Fenichel and J. C. Yochelson, "A LISP Garbage-Collector for Virtual-Memory Computer Systems," *Communications of the ACM* **12,** No. 11, 611–612 (1969).
10. A. C. McKellar and E. G. Coffman, "Organizing Matrices and Matrix Operations for Paged Memory Systems," *Communications of the ACM* **12,** No. 3, 153–164 (March 1969).
11. *IBM System/360 OS FORTRAN IV (H Extended) Compiler and Library (Mod II) Installation Reference Manual*, Form No. SC28-6861, IBM Corporation, Data Processing Division, White Plains, New York.
12. *OS ANS COBOL Version 4 Programmer's Guide*, Form No. SC28-6456, IBM Corporation, Data Processing Division, White Plains, New York.
13. *OS PL/1 Optimizing Compiler Programmer's Guide*, Form No. SC33-0006, IBM Corporation, Data Processing Division, White Plains, New York.
14. R. L. Guertin, "Programming in a Paging Environment," *Datamation* **18,** No. 2, 48–55 (February 1972).

GENERAL REFERENCES

R. V. Bergstresser, "Virtual Storage Operations," *Datamation* **19,** No. 2, 55–57 (February 1973).

L. W. Comeau, "A Study of the Effect of User Program Optimization in a Paging System," ACM Symposium on Operating Systems Principles, Gatinburg, Tennessee (1967).

P. J. Denning, "The Working Set Model for Program Behavior," *Communications of the ACM* **11,** No. 5, 323–333 (May 1968).

P. J. Denning, "Virtual Memory," *ACM Computing Surveys* **2,** No. 3, 153–189 (1970).

J. R. Martinson, *Utilization of Virtual Memory in Time Sharing System/360*, Form No. GY20-0450, IBM Corporation, Data Processing Division, White Plains, New York.

"OS/VS Programming Considerations," *IBM Installation Newsletter*, Issue No. 73-01, 2G-11G, IBM Corporation, Data Processing Division, White Plains, New York.