An experimental algorithm for optimizing program placement in virtual storage systems is described. Interprogram linkages are monitored and subsequently analyzed for frequency and proximity. The algorithm evaluates this information within the context of a paging environment. Program lists that define the optimum program placements are then generated. Performance gains are also discussed.

Optimizing program placement in virtual systems

by K. D. Ryder

As virtual storage systems[1] enjoy increasing acceptance and usage, the optimization of program execution for this environment becomes more and more important. Many facets of this general topic have already been explored, such as internal structuring of programs and function grouping based upon frequency of use.[2]

An earlier study by Hatfield and Gerald[3] presented an excellent general approach to the problem of intraprogram structure in a virtual storage environment. By way of contrast, the present study has as its genesis a very specific interprogram optimization problem related to a given software environment and derived from the development of OS/VS2. Therefore, the following discussion gives particular attention to concerns of the practicality, usability, and feasibility of the solution.

This paper describes an experimental approach to solving one particular aspect of the virtual storage optimization problem. The topic under consideration is the question of where and how programs should be placed within virtual storage, with no direct concern for either the internal structure or the data references of each program. The approach in this paper is not as sophisticated as some others currently available, nor do the experimental re-

sults indicate that performance improvements may be as great, but it does offer a simplified way of improving program execution.

Virtual storage may be characterized in many ways.[1] For the purposes of this paper, it is considered to appear as an address space that has no direct or fixed relationship to the address space represented by the actual storage size of the computer.[4] Virtual storage typically provides to the user an address space that is, in fact, larger than the storage capacity of the computer. This size disparity provides tremendous flexibility and functional capability to the user. However, it also requires that at least part of the user's address space be retained on a direct-access device to be brought into the real storage of the computer only when necessary. Clearly, the part of the user program currently executing must be in real storage.

Whenever a reference is made to part of a program that is not in real storage, the user program must be interrupted until the referenced entity is brought into the computer's storage. The unit brought into real storage is typically called a *page*, and the interruption of the user program is called a *page fault*. This terminology reflects the fact that the virtual address space is divided into fixed-size units (pages) and only those pages currently involved in program execution need to be in real storage. The overhead in time and central processor utilization for page-fault resolution represents a principal cost factor in virtual storage systems. Therefore, most efforts to optimize use of the virtual storage environment address this problem in one way or another.

It is obvious that a major source of page faults can be the linkage of one program to another. For example, if program A passes control to program B, a page fault will occur unless program B is in real storage. This paper investigates an algorithm for ensuring that such page faults are held to a minimum. Optimization to minimize other sources of page faults, although a broad and significant topic, is not considered here. The algorithm will be discussed in terms of its implementation within a single, specific virtual storage system—the IBM operating system OS/VS2.[4-7] The implementation was within the context of the first release of OS/VS2, that is, OS/VS2.1. However, the general concept and algorithm have a direct application in any virtual storage system.

A significant feature of the OS/VS2.1 system is the link pack area (LPA). This consists of a portion of virtual address space, normally one to two megabytes in size, which contains both system- and user-supplied programs. Due to the large number and variety of programs within the LPA, one may expect the bulk of program linkages to involve programs within the LPA. Accordingly, it was decided that the prime objective would be the optimum placement of programs within the LPA. This placement, or pack-

Figure 1 LPA phase output

Program Name	*Size*	*Starting Virtual Storage Address of Program*
Program 1	Size 1	Address 1
Program 2	Size 2	Address 2
Program 3	Size 3	Address 3
Program 4	Size 4	Address 4
. . .	. . .	. . .

(1 entry per program in the LPA)

aging, would minimize the incidence of page faults related to interprogram linkages.

Algorithm operation

The process of optimizing the LPA structure may be logically divided into three phases. The first phase, the LPA phase, determines the LPA structure as it currently exists. The key information includes the name, size, and location of each program. This LPA data is then stored for later reference (see Figure 1).

The second phase, the trace phase, monitors the linkage activity of the system as it is operating with a normal work load. During this tracing activity, the only deviation from a purely normal user environment is the operation of the trace monitor itself. The function of this monitor is quite simple and therefore presents a very minimal interference factor. The user environment has total flexibility—it may be multiprogrammed, batch, interactive, or any combination thereof. The user environment should accurately represent the operating environment to be optimized.

As program linkages are established, the trace monitor notes the task that is attempting the linkage and the address to which linkage is being established (see Figure 2). This information will be put to use in the final phase of the algorithm's operation. In terms of the OS/VS2.1 implementation, the linkages to be traced are identified via LINK, LOAD, XCTL, and SYNCH supervisor calls (SVCs), Error Recovery Program (ERP) invocations, and issuances of type 3 and 4 SVCs.[7] The key consideration is that all of these linkages require operating system assistance and are,

Figure 2 Trace output

Task I.D.	*Virtual Storage Address of Linkage*
Task 1	Address 5
Task 1	Address 9
Task 3	Address 6
Task 7	Address 11
Task 1	Address 3
Task 1	Address 20
Task 2	Address 8
Task 1	Address 2
. . .	. . .

Time

(1 entry per monitored linkage)

therefore, easily monitored. In addition, such monitoring will normally encompass all the interprogram linkages within the LPA.

The third and final phase of the algorithm is the pack phase. In this phase, the output of the previous phases is analyzed, and an optimum LPA structure is developed. The output of this final phase consists of sets of program names, indicating which programs are to be placed together in a page (or pages) of virtual storage.

linkage affinity

The data collected during the trace phase is purely sequential, and it must undergo additional processing before it can be used. Typically, the trace data at this point reflects the activity within a multiprogramming environment. To derive usable linkage sequences, it is necessary to sort the trace data on a task basis. Thus, the final form of the input is multiple, disjoint sets of linkage data, time-ordered on a task basis (see Figure 3). These sets may now be analyzed, selecting the sets in any order, to understand the linkage affinity that one program has for another.

The concept of a linkage affinity is fundamental to utilization of the trace data in the pack phase. For a given task, the trace data

Figure 3 Trace output sorted by task

Task I.D.	*Virtual Storage Address of Linkage*
Task 1	Address 5
Task 1	Address 9
Task 1	Address 3
Task 1	Address 20
Task 1	Address 2
Task 1	. . .
Task 1	
Task 1	
Task 1	
Task 1	
Task 1	
Task 1	

Time ↓

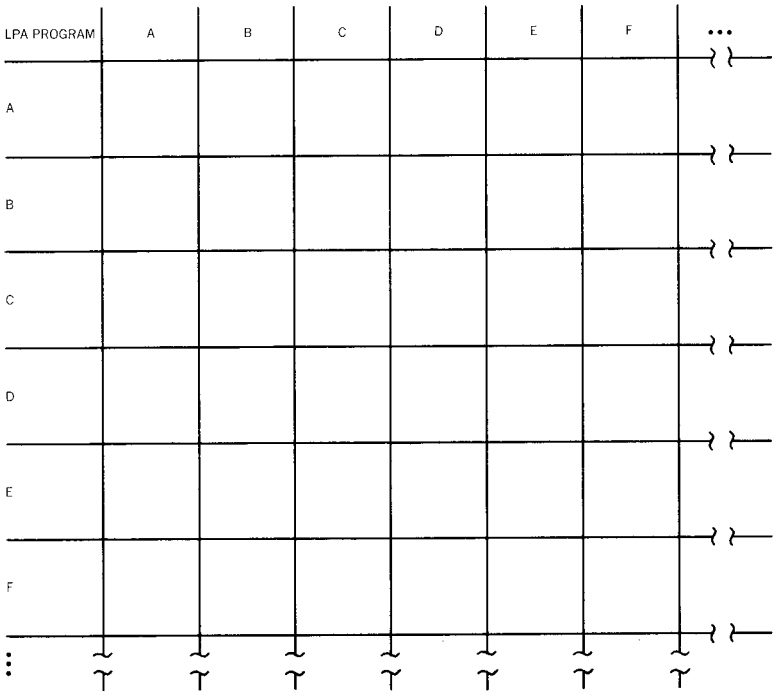
(1 Table per Task)

may show the following linkage sequence: Program A ⟶ Program B ⟶ Program C ⟶ Program D ⟶ Program E ⟶ Program F ⟶ . . . It is clear that Program A has a linkage affinity for Program B. It is also true that some lesser affinity exists between Program A and Program C, between Program A and Program D, etc. For simplicity, the affinity of Program A to Program B will henceforth be denoted as (A, B). For the OS/VS2.1 implementation of the algorithm, affinity values were assigned as follows:

$$(A, B) = 5;\ (A, C) = 4;\ (A, D) = 3;\ (A, E) = 2;\ (A, F) = 1.$$

It can be seen that interprogram affinities were considered only to a level of five, i.e., (A, G), (A, H), and so forth are insignificant. The actual affinity values chosen were equally arbitrary, but they proved to be workable values. Clearly, the set of values that was used is among the simplest possible combinations, and the values were chosen primarily for their simplicity. The favorable results obtained with the algorithm may not reflect its full potential since only limited experimentation was done with alternative values. The levels of affinity to be considered and the affinity values selected in no way detract from the generality of the algorithm.

Figure 4 Affinity matrix



Given that an affinity value may be assigned to pairs of programs, it is now necessary to accumulate these values for each program pair over the period that was monitored.

affinity matrix

The logical mechanism by which this is achieved is the affinity matrix. This matrix has a row and a column for each program that exists in the LPA (see Figure 4). As the task-related sets of trace data are now examined, the appropriate elements of the matrix are incremented. Thus, for the sample linkage sequence Program A ⟶ Program B ⟶ Program C ⟶ Program D ⟶ Program E ⟶ Program F ⟶ . . ., matrix elements would be incremented as follows:

(A, B) incremented by 5

(A, C) incremented by 4

(A, D) incremented by 3

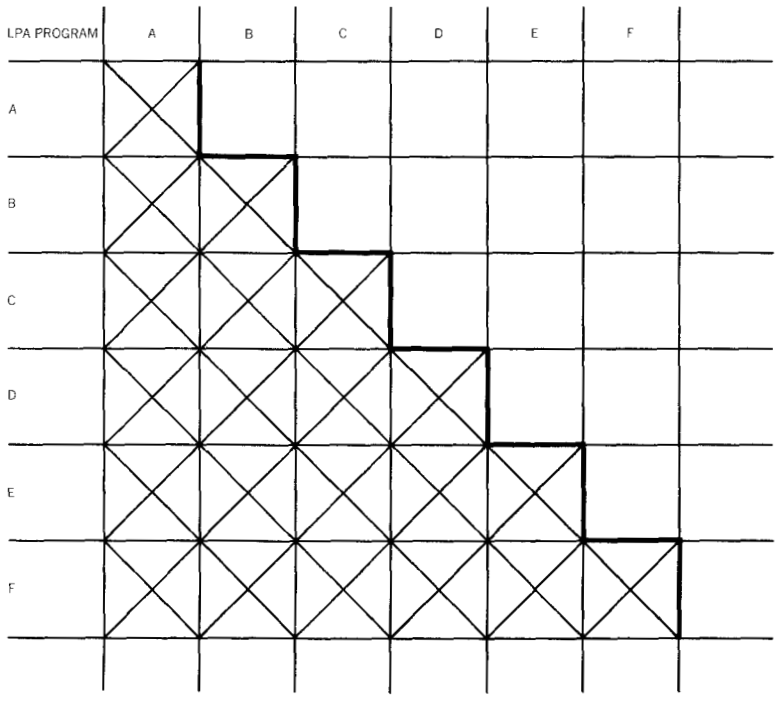
(A, E) incremented by 2

(A, F) incremented by 1

(B, C) incremented by 5

(B, D) incremented by 4

Figure 5 Affinity submatrix



If a given program in a linkage sequence does not reside in the LPA, any couplet of which it is a part is simply disregarded. Hence, if Program C were not in the LPA in the above example, no matrix element involving C would exist. All other increment values would remain unchanged. All trace data involving non-LPA programs is thus purged as the affinity matrix is built.

An obvious simplification of the affinity matrix is to (1) discard all entries of the form (A, A) since the affinity of a program for itself is of no interest and (2) combine all entries of the form (B, A) with (A, B) since it makes no difference whether A establishes a linkage to B or vice versa. In this latter case, only the frequency and proximity of the A and B linkages are important.

An actual implementation of the algorithm would normally never build the full affinity matrix, although it has been described here for generality. The resultant submatrix is shown in Figure 5.

Before the actual preparation of program packing lists, the submatrix must be simplified further. This simplification process recognizes several fundamental considerations. These considerations reflect the requirements of the OS/VS2.1 environment. The rules should also apply in large part to any virtual storage situation.

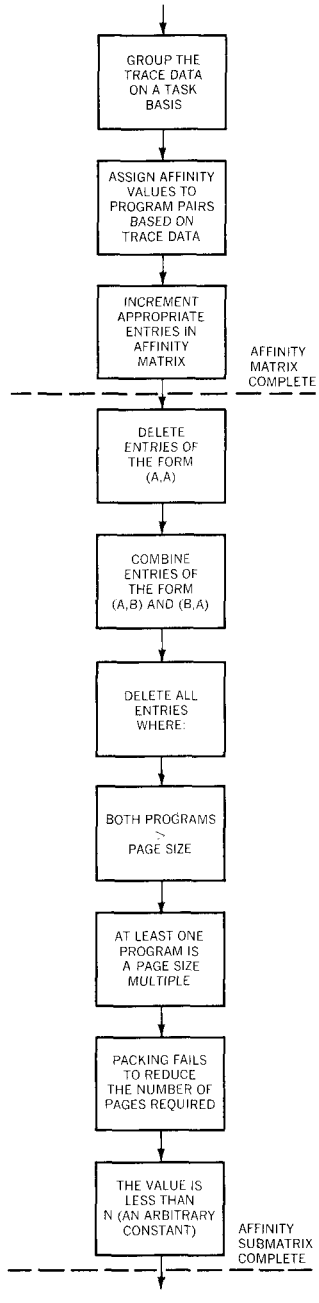
1. If a program is greater than a page in size, then it should be positioned on a page boundary. This rule allows the programmer some capability to anticipate page faults and to structure his program internally in an optimum fashion. Otherwise, the point in the program at which a page boundary is crossed will be totally unpredictable to the program. The implication of this requirement is that no affinities should be considered between programs when both are larger than a page in size. Obviously, such a combination would not reduce the number of pages required for the two programs if both programs are constrained to a page boundary. The corresponding entries in the submatrix are therefore deleted.
2. If any program is an exact multiple of the page size, then it cannot be combined with any other program within a page. Accordingly, all matrix entries involving such programs are deleted.
3. It is allowable to combine a program greater than a page in size with one of less than page size. However, for any two programs, their combination in a single page (or pages) must require fewer pages than they would require separately. For example, a program requiring one and three fourths pages should never be combined with one requiring three fourths of a page. To do so would involve three pages, exactly what would be required if no effort at combination was made.

   Furthermore, program placement in the LPA should be such that no program spans a page boundary unless its size is in fact greater than a page. To violate this assumption would greatly increase the page-fault rate during execution of such programs. Not only would the program be highly susceptible to page faults, but it would also be unable to predict the point within the program at which the page boundary would fall. The program structure, therefore, could not be optimized around the location of this boundary. Phrased in different terms, this consideration states that for any two programs, their individual sizes modulo page size may not produce a sum that exceeds a page in size. The affinity submatrix is, therefore, purged of any such combinations.

4. If the value of any entry in the affinity submatrix is less than an arbitrary value, it is deleted. For the purposes of this implementation, that arbitrary value was zero. That is, all remaining submatrix elements were considered. Figure 6 summarizes the key operations in first constructing the affinity matrix and then deriving the affinity submatrix.

It should be pointed out that the restrictive nature of these rules does not in fact preclude dense packing of the LPA. Even though some affinities cannot be translated into program packing lists, other affinities will normally exist and can be used to fill out

Figure 6 Building of matrix



each list. Also, as will be noted later, when the actual LPA is constructed, the affinity-based packing of the LPA is followed by a space reclamation phase that effectively eliminates any unused spaces.

final stages

The algorithm now enters an iterative process whereby the affinity submatrix is searched and program packing lists are developed. The largest value in the affinity submatrix is now located and deleted. Either of two situations may prevail:

1. Neither of the programs associated with the selected matrix element has yet been added to a packing list. This is obviously the situation on the initial scan of the matrix.
2. (a) both programs are already in packing lists or (b) one and only one program is already in a packing list.

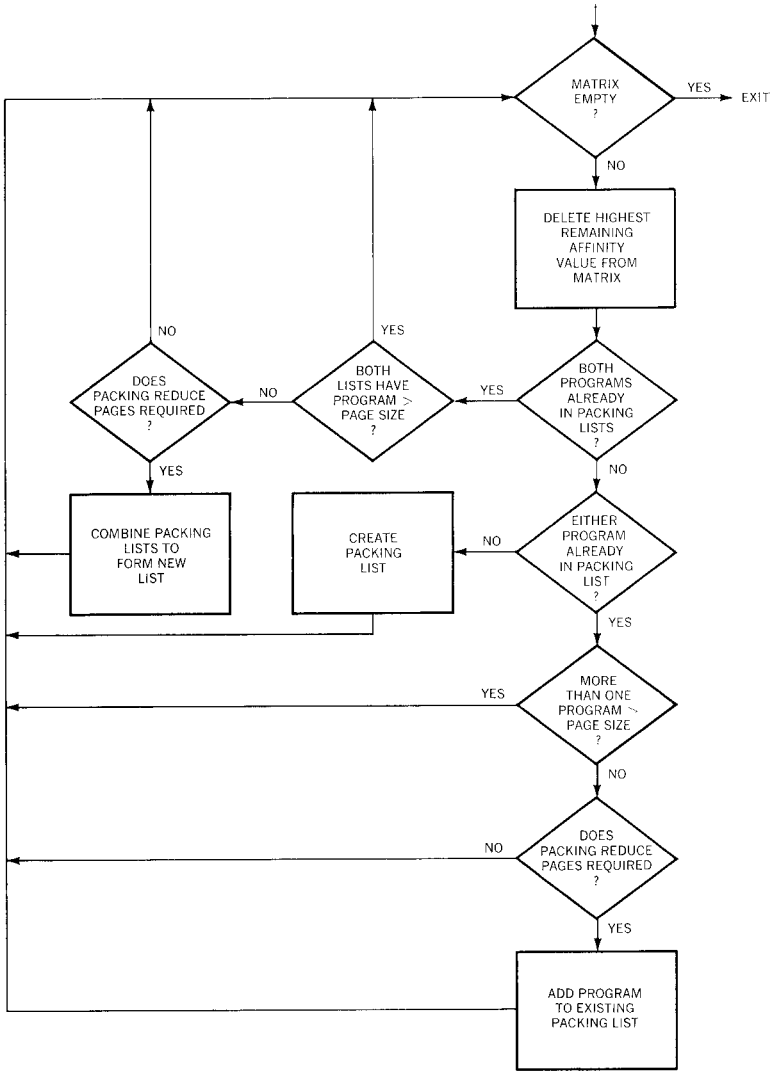
In Case 1, processing is fairly straightforward. The earlier purges of the affinity matrix have guaranteed that the program pair can, in fact, be packed successfully in one or more pages. Hence, a program packing list is established for these two programs. Also, the space remaining in the page (or pages) to be occupied by the programs in this packing list is calculated. This calculation is performed by taking the program sizes modulo the page size, summing these values, and subtracting the sum from the page size. In the event that this remaining space is smaller than any as-yet-unpacked LPA program, then this program packing list is complete. If the list is, in fact, complete, matrix elements involving elements of this list are purged to avoid their further consideration.

Case 2 represents the more complex situation. It may be viewed more simply by considering two subcases. In Case 2a, both programs have already been included on program packing lists. All matrix interactions between members of the two program lists are now deleted. No benefit will result from future considerations of affinities between elements of the lists. If both lists contain programs that are greater than a page in size, then no combination is possible, as discussed earlier.

A further test is required before the two program lists can be merged. Just as noted earlier for two *programs*, it is not desirable to combine *program lists* unless they require fewer pages together than singly. That is, the sizes of the two program lists modulo page size should not yield a sum greater than page size.

If the preceding tests are met successfully, the two lists are merged to form a single program packing list. The space remaining in the page (or pages) to be occupied by the members of this list is now recalculated. As before, a space remainder smaller

Figure 7 Packing list creation



than any still-unpacked LPA program will indicate a completed list. Matrix elements involving members of the list are deleted once the list is completed.

For Case 2b, it is necessary that one and only one of the two programs already exist in a packing list. To simplify future processing of the affinity matrix, all matrix interactions between the unassociated program and programs in the list are purged.

The addition of the selected program to the existing list may cause the list to contain multiple programs larger than a page in size. For the reasons discussed earlier, this situation is undesirable, and no such combination is allowed.

The final test requires that the unassociated program fit in the page(s) to be occupied by the list members. Note that the program may be larger than a page, in which case its size modulo the page size is considered.

As usual, the addition of the unassociated program to the list causes a recalculation of the space remaining in the page(s) associated with the list.

resultant LPA

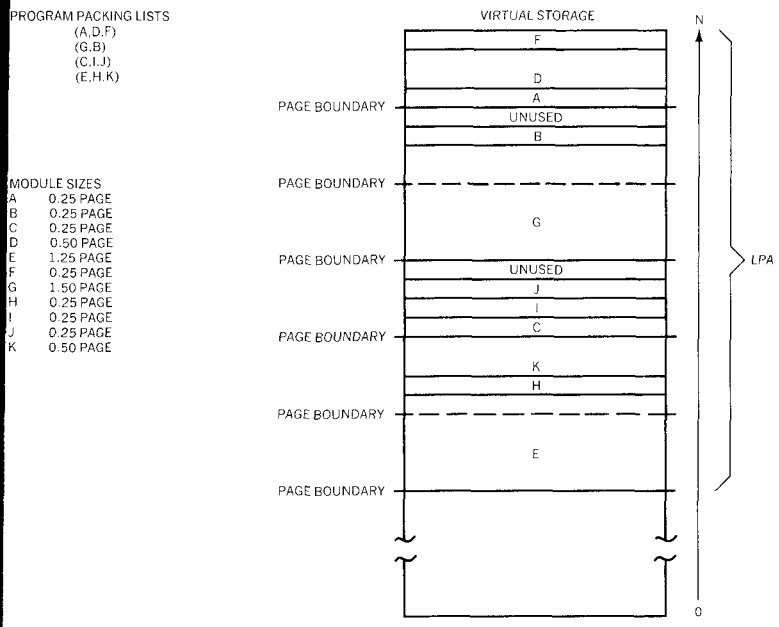
The algorithm continues to cycle through the affinity matrix until all entries have been deleted. The end product is a set of program packing lists that may be used to optimize the placement of LPA modules in virtual storage. Each list is an entity and may be considered in any order relative to the other lists. For the OS/VS2.1 implementation, the format of these lists matches exactly that of the lists used to build the LPA at system initialization time.[5, 6] Hence, the algorithm output is simply and directly used to structure the LPA. The algorithm operation of actually building the packing lists is summarized in Figure 7. Only the key functions are shown.

In Figure 8, the resultant LPA can be seen, given a particular set of program packing lists. It should be noted that, for convenience, any program greater than a page in size appears as the first element in the list. This facilitates its placement on a page boundary, as shown. In the OS/VS2.1 environment, the loading of programs named in a list is done in the order of their appearance in that list. The first program is assigned the lowest virtual address in the page or block of pages for that list, and succeeding programs in the list are assigned progressively higher addresses. Each successive list is associated with a progressively lower block of address space.

Another notable feature of the OS/VS2.1 implementation involves those LPA programs for which no significant affinities were recorded. Such programs are used to fill in the unused spaces in the LPA. The assignment of available space is made on the basis of program size, with the largest programs considered first. This process allows efficient space utilization in conjunction with affinity-based program packing.

It is reasonable to assume that variations in work loads would tend to involve different sets of LPA programs but would not drastically affect the linkage patterns between particular programs. In this sense, the packing lists generated from different trace runs would be additions to, rather than replacements for, existing lists. The OS/VS2.1 implementation adopted a flexible approach in this area, however, by allowing for the definition of multiple sets of program packing lists. If it is determined that different work loads generate significantly distinct LPA struc-

Figure 8 LPA structuring based on program packing lists



tures, then the appropriate set of program packing lists can be assigned the proper name to cause its selection at system initialization time.

Performance measurements

The algorithm has been subjected to formal measurements and has demonstrated significant performance gains in an OS/VS2.1 environment. Measurements were made on a System/370, Model 155-II. Work loads were used that were made up of COBOL, FORTRAN, SORT, and Basic Assembly Language (BAL) programs. Various jobs included assembly, compilation, linkage editing, and/or execution steps. Each work load required approximately 10 minutes for completion in the environment established for the measurements.

The average performance improvements are summarized in Table 1. The gains realized for each of the work loads were quite similar. Comparisons are relative to the same system with an LPA that was not affinity-packed. The table shows the effect of an affinity-packed LPA on a number of key performance parameters. Supervisor state CPU time reflects those periods when the CPU is not in a waiting condition and is not executing in the problem state. Supervisor state normally is associated with control program execution. As shown by the table, this overhead was decreased by seven percent.

Table 1 Performance improvements

Parameter	*Change with packed LPA*
Supervisor state execution time	7%
Total run time	5%
I/O interruptions	10%
Paging channel time	21%

The total run times for the work loads were decreased by an average of five percent. The total number of I/O interruptions, which is influenced by the incidence of paging operations, decreased by 10 percent. This can be attributed directly to a decrease in the number of paging operations required. Similar considerations apply with regard to the 21 percent reduction in utilization of the paging channel. This latter savings is especially significant for configurations in which the paging channel is shared with other I/O operations.

Although these results were obtained in a formal measurement environment, they are not presented as an exhaustive analysis of the algorithm's capabilities. Rather, these results should be regarded as a statement of the potential of this approach to program packing. Direct performance comparisons to other optimization techniques should not be rigorously applied due to the many variables affecting such performance figures. Absolute comparisons are reliable only when factors such as work load, real storage, hardware and software configurations, etc. are strictly controlled and directly comparable.

In these measurements, no attempt was made to induce an artifically high, or even heavy, paging load. The base system for measurement comparisons experienced less than 20 paging operations per second. One would suspect that performance gains would be even more substantial as the paging rate increases. To some extent, this contention has been supported. A small OS/VS2.1 system, using the algorithm experimentally, reported a 25 percent decrease in work load run times. Since the small system was characterized by a high ratio of virtual storage to real storage, it would normally experience higher paging rates. It would, therefore, be favorably affected by proper packing of the LPA.

Summary comment

Experience with the algorithm to date indicates it to be a worthwhile step toward optimization for a virtual storage environ-

ment. The algorithm is clearly not the ultimate solution to program packing, yet it does provide an automated and effective means to this end. As such, it has proved useful and far preferable to manual or intuitive techniques for program placement.

The generality of the algorithm is such that it may readily be applied to any virtual storage environment. The details of the LPA, trace, and pack phases may change for a given environment, but functionally they are still applicable. The pack phase is, obviously, the key element in the overall process. The algorithm implicit within this phase is essentially independent of both the LPA and trace phases. The output of the LPA phase is simply a table of virtual addresses and is not uniquely nor intrinsically related to the LPA. Likewise, the trace phase produces a sequence of virtual addresses that can be gathered in any number of ways and that need not be restricted simply to program linkages.

The low overhead and simplified assumptions inherent in this implementation suggest that it could readily provide a tool for use during normal system operation. As such, the algorithm would allow dynamic system monitoring in any installation. Updated program packing lists could then be generated periodically, as required.

One possible extension of the algorithm could be applied to branch linkages. In this case, the LPA phase would consist of building (or predefining) a directory of the programs to be monitored. The trace phase could simply monitor the linkage points under consideration. This could be readily achieved via temporary modifications to the linkage points. Each linkage point could then invoke the tracing subroutine. The pack phase would operate essentially unchanged, with the format of the packing lists being produced in whatever form is most usable.

This analysis of interprogram linkages does not address the significant area of data references. Since such references represent a source of page faults, their consideration would provide an additional input to the optimization process. Obviously, practical difficulties arise in attempting to determine what these data reference patterns are. The normal approaches to collecting such information imply more sophisticated tracing techniques, much greater volumes of data, and substantially more overhead, compared to the simple linkage trace described here.

Even without adapting the algorithm to new applications or additional inputs, much additional information could be derived form further measurement and analysis. For example, attempts could be made to optimize the assignment of affinity values, as opposed to the linear 5-4-3-2-1 approach now used. The depth

to which affinities should be considered is also an open question; the present value of five is a workable, but possibly not optimum, value. There is also the question of the minimum allowable value in the affinity matrix. The present implementation considers any nonzero value in the matrix. It may be, however, that nonzero values below a certain minimum should be disregarded. Low-affinity programs would then be packed on the basis of optimum space utilization, not affinity.

Finally, there is an additional level of complexity that can be introduced to the pack phase. As described in this paper, the pack phase repeatedly scans the affinity submatrix and deletes the highest value found. This matrix element identifies two programs that are then placed in the same packing list, if possible. This straightforward approach becomes more complex if one attempts to consider groups of matrix elements rather than single elements. For example, (A, B) may represent the highest value in the affinity matrix. Yet it may be that packing A with C, D, and E will yield superior results even though (A, C), (A, D), and (A, E) are all individually less than (A, B). Such a result is at least theoretically possible.

In conclusion, it can be said that the algorithm offers tangible benefits in its present form and provides a basis for further experimental investigation of the virtual storage environment.

ACKNOWLEDGMENTS

A statement of appreciation goes to those who made significant contributions to the implementation of the algorithm: D. E. Coon, who developed an initial APL version; A. F. Banks, who developed the full-function Basic Assembly Language (BAL) version; R. H. Van Dam, who tested the BAL version; P. A. Chiappa, A. G. Jaeger, E. T. Boyle, and I. H. Schneider, who made many helpful comments and suggestions during the design of the program.

CITED REFERENCES

1. P. J. Denning, "Virtual memory," *Computing Surveys* **2,** No. 3, 153–189 (September 1970).
2. D. Ferrari, "A tool for automatic program restructuring," *Proceedings of the ACM 1973 Annual Conference*, 228–231 (1973).
3. D. J. Hatfield and J. Gerald, "Program restructuring for virtual memory," *IBM Systems Journal* **10,** No. 3, 168–192 (1971).
4. *Introduction to Virtual Storage in System/370*, Form No. GR20-4260, IBM Corporation, Data Processing Division, White Plains, New York.
5. *OS/VS2 Planning and Use Guide*, Form No. GC28-0600, IBM Corporation, Data Processing Division, White Plains, New York.
6. *OS/VS2 IPL and NIP Logic*, Form No. SY27-7243, IBM Corporation, Data Processing Division, White Plains, New York.
7. *OS/VS2 Supervisor Logic*, Form No. SY27-7244, IBM Corporation, Data Processing Division, White Plains, New York.