trends in
DATA ORGANIZATION
AND
ACCESS METHODS

P. C. Patton
Analysts International Corporation
Minneapolis, Minnesota

There has been a trend in applications programming over the past fifteen years from highly structured programs and loosely structured data toward loosely structured programs and highly structured data. Transaction oriented systems and other data directed programs make greater demands on the memory hierarchy of a computer system and have given rise to the need for storage media which are to some degree structurable under program control. Overhead in operating systems and large on-line application programs is described as a symptom of "static" and "dynamic" mismatch problems in the storage hierarchy. Some new memory organization techniques beginning to appear on computers may allow greater efficiency in data directed systems since they allow the programmer to store the structure and the content of his data separately.
Application programming developments and trends

Over the past fifteen years there has been a trend in application programming from highly structured programs and loosely structured data toward loosely structured programs and highly structured data. In a large measure this trend was made possible by the transition from magnetic tape as a secondary storage medium to random access mass storage media and large scale drums. On the UNIVAC I the processing of a tape file to develop a list-of-materials or where-used-list from a parts catalogue could take several hours, since a complex program running in many overlays would have to perform many tape passes. On a later machine with a drum or disc capable of holding the entire file this problem would take only minutes. Hardware developments have made available many possibilities for programming sophistication and programmers have always been quick to seize on them.

Some application programming techniques which exemplify the trend described here are data directed programs and transaction oriented systems. These examples are more a difference of degree than kind, but serve to illustrate the trend. A data directed program is a loosely or variably structured program, or sometimes just a set of functional subroutines. The linkages between these subroutines are more dependent on data characteristics and requirements than any hierarchical structure expressed within the program itself. Such programs show a degree of self-adaptivity in that they are able to organize themselves to perform in such a way that they achieve a certain end in processing the data, yet their path toward that end cannot be readily determined. Heuristic programming, learning programs and other non-deterministic simulation approaches are data directed programs whose structure is dependent to some degree on the structure inherent in the data they process.

Transaction oriented systems and application programs are similarly data directed, but the data they process is usually divided into two classes; i.e. such systems process a large number of relatively small data packets or data sets against a very large highly structured data bank or database. From one instant to the next, one cannot predict deterministically the activity of the various functional subroutines making up the system, since these activities are dependent on the many transaction data packets arriving in real-time. The operational activities are also dependent on the structure and content of the database and the changes being made in it by the results of transaction processing.

By giving the programmer large random access mass storage media, the memory designer has opened Pandora’s box. Programmers have developed systems employing storage hierarchies to store their data hierarchies, but in many cases their designs seem to have grown out of scale with the capability of the hardware. We all know of large scale hardware systems which theoretically can add, subtract, fetch and store rapidly enough to complete all the processing necessary to do the job, yet these systems fail to perform adequately when put on-line. Why? Overhead often takes the blame, but overhead is just the symptom of mismatch between the data hierarchy and the memory hierarchy (the "static" problem) and perhaps also between the secondary store transfer rate and the main store transfer (the "dynamic" problem).

Before exploring these data storage and data transfer problems, some modern day data organization methods should be reviewed.

Highly Structured Data

Linear Lists: The simplest organization of a group of data values or data sets is the linear list, yet here already there are many possibilities for allowing insertions and deletions to such a simple linear list. Gaining access to the nth node of a long list for an arbitrary value of n is not simple if insertions and deletions are being made throughout the list. To simplify list structures and their use one usually requires that insertion and deletion are made only at one or both ends of the list. Some of the most commonly encountered structures are shown in Figure 1. These are:

The stack — a linear list in which all insertions and deletions (and usually all accesses) are made at one end of the list. The stack is often referred to as a LIFO (Last In, First Out) list or a push-down list. It is commonly used in such applications as processing natural or artificial (compiler) languages.

The queue is a linear list in which all insertions are made
at one end of the list and all deletions (and usually all accesses) are made at the other end. The queue is sometimes called a FIFO (First In, First Out) list and is found in such applications as executive systems, message switching systems and input/output handlers.

The deque is a linear list in which all insertions and deletions (and usually all accesses) are made at the ends of the list. The deque is also referred to as a double-ended queue and may occur in the form of output-restricted or input-restricted deques, in which deletions or insertions occur only at one end or the other.

A list may be allocated in memory sequentially or it may be stored in such a way that it is linked; that is, each data set contains a partial word address, link or pointer to the next data set. Linked storage allocation of structured lists have been called the "threaded list" technique; however, this terminology as originally proposed by A. J. Perlis and C. Thornton was somewhat more specific than general linked allocation.[4] Note that a list whose final data set or node, points to its first is a circular linked list. Another feature often found useful in simulation applications is the doubly linked list, in which each node points not only to its successor but its predecessor as well. Such list structures are called symmetric lists and the FORTRAN based list processing language SLIP developed by Welzenbaum is a popular one for processing them.[4]

Arrays and Orthogonal Lists: The simplest generalization of a linear list is a two-dimensional (or higher dimensional) array of information. The most common example is the \( m \times n \) matrix with \( m \) rows and \( n \) columns of sequentially stored data values. It is uncommon to encounter up to five dimensional tables of information in physical problems, where for example the indices may be considered to be rows, columns, pages, volumes and tomes.

Even simple arrays can raise interesting problems in allocation sequence or linking and efficient retrieval index calculation, particularly if only part of the matrix, say the upper or lower triangle (or tetrahedron for higher dimensional arrays) is to be stored. These problems become particularly acute when the matrices are sparsely populated such as those which occur in linear programming and aircraft structures problems.[4] As an example, the author once aided in the analysis of a F-104-G Superstarfighter by lumping mass and elasticity values to every fourth structural member. Even this rough grid on such a small airplane produced mass and stiffness matrices of order 10,000 by 10,000. Although these matrices were populated to less than 1%, the problem of inverting one of them, multiplying it by the other and then finding the first 150 eigenmodes of the product proved too complex to handle on a very large computer. The number of actual multiplications and additions was not overwhelming, but the structural and data organization problems made the task infeasible.

**Tree Structured Data:** A powerful method for structuring data is the tree. A tree is a finite set of one or more nodes, one of which is designated the root of the tree and the remaining nodes are partitioned into disjoint sets each of which is in turn a tree. The nodes are, of course, joined by branches. Some examples of data trees are family trees such as the pedigree, which traces from an individual backward toward his ancestors in the form of a binary tree. The linear chart is the opposite of a binary tree, since it traces from a given ancestor forward toward all of his descendents. Some examples of trees are shown in Figure 2.

Tree structured data is useful in many applications. One with which the author is most familiar is storing sparsely populated matrices as trees. A matrix organized in this manner is called a hyper-matrix, since it is a matrix whose elements are matrices which in turn have matrix elements on down to \( n \) levels.[4] These matrices arise naturally in structural engineering and allow one to store enormous, but sparsely populated, matrices without storing all the zeros. For greater storage efficiency one tends to separate the structural information from the data values or content. Using a large storage area to store a matrix most of whose elements are zero is inefficient, since it wastes vast amounts of data storage to record simple structural relationships.

Linked list allocation techniques can be applied to trees to produce more complex data structures. Figure 3 shows a ring structure which consists of circular linked trees. Symmetric linked trees with \( n \) links per node are an ex-
ample of a very complex extension of this simple structure. The directed graph is a more complex extension of a tree, but these data structures find extensive application in transportation networks, electrical networks, planning methods and many others. A very complex data structure is the coral type structure which may be visualized as data sets represented by beads, each having one or more holes. The beads are then strung by one or more threads. There is an interpretative list processing language called CORAL for processing such data.

This trend toward more complex data structures in both systems and application programming may be highlighted by directing attention to the file management problem. Files are often complex tree structured data and most file processing today is done in COBOL. Now COBOL is a good language, but it is not readily able to communicate with today's complex executive and operating systems in such a way that files are conveniently managed by the application programmer. To aid this deficiency a file management system is sometimes sandwiched between the COBOL generated object program and the operating system. This files management system then manages data stored in a hierarchy of storage media. Some familiar file management and information management systems are IMRADS, GIS, XMIS, IDS, not all of which have achieved full implementation. The latter, Integrated Data Store, allows the COBOL programmer of a General Electric 400 or 600 Series computer to define simple tree structured file relationships in his File Definitions; during compilation, code is generated which will later guide the operating system in the storage and retrieval of this data.

Such file management systems are at about the interpretative programming level and are extremely inefficient, leaving much to be desired. To date no software or interpretive approach to this problem has proved notably successful.

Memory Organization

A representation of a simple storage hierarchy in a present day computer system is given in Figure 4. It ignores such complex features as multiprocessing and multi-accessing in order to focus clearly on the memory system problem at hand. The various nomenclature of the system designer, the application system programmer and that of the hardware designer are given as well. As the Figure shows, in most systems today the secondary and tertiary storage media are accessed by means of the input/output subsystem or logic. This situation leads to most large scale applications programs being input/output bound at the secondary storage level. The more highly structured the data, the greater any inherent latency at this interface aggravates this problem. Sophisticated programming may tend to eliminate the effects of latency at this interface only if the process involved is deterministic and well known. On a data directed or transaction oriented system with complex data structures, one can at best aim at a statistical solution to this problem during program design.

A system organization technique which may help to relieve this particular problem is shown as a double line on the left of the diagram. If secondary storage can be accessed directly and main core to/from bulk core (or
drum) transfers can be made without I/O control, then the system is said to have an integral secondary store. This technique has appeared in some recent machines and is designed into machines now appearing. It will certainly be helpful in reducing the overhead resulting from secondary to primary transfer mismatch in many applications.

If secondary storage is word addressable and integral, and if instructions can also be executed out of this integral secondary store, then many more possibilities are available to the programmer for reducing the mismatch for the application program or operating system as well as the data. Of course, in this case he must develop a dynamic algorithm which decides when to execute a group of instructions in secondary store and when to transfer them to primary store for execution. The IBM 360 Series allows bulk core to be connected in an integral fashion such that data can be reached either directly or via the I/O section. The DATAC 110 computer has an elegant primary/secondary memory approach plus a variable field table-look-up instruction for locating data in the secondary store (disc). However, this disc is fully integral and is not reached by the I/O section. On such a core/disc machine (core/drum, as the earlier UNIVAC 1100 series), the decision of whether to execute out of core or disc/drum is easier than it would be if the secondary store were bulk core only slightly slower than the primary core store.

Basically what the programmer wants in a large multilevel storage hierarchy is the capability for the hierarchy to appear as a single-level store. There are several software techniques which achieve this by direct simulation; however, not with notable efficiency, and there are also a few hardware techniques, all of which have the same failing. One of the first machines which tried to simulate a large single level store with a storage hierarchy was the ATLAS I, which used a core/drum hierarchy and had a paging algorithm which employed a hardware learning scheme to make swapping decisions. This technique was frankly admitted to be necessary because a large primary core was too expensive. Since that time, it seems that system and software designers have gotten so involved in the problems of paging, swapping, overhead, virtual memories, etc., that it may have been overlooked that these are technological headstands to avoid an economic problem — the high cost of fast core memories for large primary stores.

The memory system lies at the heart of the computer system and any attempt to solve the system organization problem always resolves itself to a memory system design problem. One might state the system organization problem as: "How can one gain an order of magnitude in computer system capability by organization factors alone." Most of the attempted solutions to this problem to date have become part of the problem because of their associated "overhead."

What might the user require during the next decade

To close on a more optimistic and hopeful note, some memory techniques that have attracted the attention of users and which might find more use in forthcoming machines might be mentioned. It seems evident that the programmer would be aided by a hardware capability to structure the memory of the computer under program control. In general this problem sounds unsurmountable, but one can quote a few specific cases. An obvious one is the language machine, i.e. a computer whose repertoire is a high level language like FORTRAN, ALGOL or COBOL or even a straightforward transform of one of these. The Burroughs 5500 and 6500 are essentially ALGOL machines; their main memory is organized as a push-down list or stack. This organization is not fully general since a stack is a particular type of linear list, but the stack is a very useful list organization.

Another specific example is the ATRAN string processor which allows the programmer to divide memory into data, buffer, and program areas and then to address the data area by an R/I or record/item addressing scheme. Such an addressing structure is ideal for message switching, programmable control units and many business applications. A useful extension of this technique to an entire hierarchy would be:

R/I record/item addressing in primary store,
F/R file/record addressing in secondary store,
A/F application/file addressing in tertiary store.
If such a machine allowed the programmer to structure memory dynamically by program control, then the integrated data store problem would be solved. This machine would essentially be a COBOL computer with a hardware file management system. Variable structure memories would allow the programmer to match the memory structure and thus solve the static mismatch problem.

Another system organization technique which may allow the achievement of an order of magnitude improvement in performance is logic-in-memory. Logic-in-memory and cache memory techniques are a solution of the dynamic mismatch problem between the primary store and the processor. Similar techniques can be employed to solve the primary store to secondary store mismatch; however, the long word length secondary store as used in the Control Data Extended Core Store (ECS) is probably a more economical solution to this problem. The practicality of logic-in-memory technology is largely an economic problem and depends almost entirely on the advancement of microelectronics technology and the development of new algorithms that make good use of their power.[8]

Finally, one might address the question that is probably in every hardware and system designer's mind after hearing these ideas. "Why should we stand on our heads to solve what is a troublesome programming problem?" This question can be answered by pointing to another trend over the past 15 or 20 years: What were originally application techniques to solve a programming problem soon became interpretive coding schemes and were eventually cast into hardware. One might choose two examples: one data processing, one scientific computation, to illustrate this trend. In the early days of computing the handling of alphabetic data required first developing an encoding scheme and then writing tables and programs to deal with encoded alphabetic data as numerics. Now computers handle alphabetic data automatically in hardware and some have more than one built-in character code. The first matrix computations dealt with carefully scaled data which was processed in fixed point arithmetic. Soon floating point interpretive schemes were common, quickly followed by floating point hardware on the IBM 704 and UNIVAC 1103A. Many scientific machines today even have double precision floating point hardware.

Today's data processing application programmers take alphanumeric character encoding for granted, but they are struggling with highly structured files and complex data bases. Today's scientific applications programmer takes floating point arithmetic for granted, but he would like to be able to store a sparsely populated 10,000 x 10,000 matrix without storing all the zeroes. In short, there is a need for storage media which allows one to store the structure and the content of the data separately.

Acknowledgments

The author wishes to acknowledge discussions with Dr. R. H. Fuller, the panel moderator, with the other panelists and with Messrs. V. C. Benda, W. P. Burrell and J. O. Ross.

REFERENCES