Variable topology random access memory organizations

by MARTIN A. FISCHLER and ALLEN REITER

Lockheed Palo Alto Research Laboratory
Palo Alto, California

INTRODUCTION: THE BASIC CONCEPT

One of the most basic of all computer operations is the actual or virtual construction of a data sequence to be used either as the operand for a simple data transfer, or as the argument of a functional transformation. In a significant number of practical situations, the data from which the string is to be constructed are physically scattered prior to the proposed operation or must be scattered after the operation due to physical space limitations or for reasons dictated by the logical structuring requirements of the application.

Powerful software schemes (e.g., the list processing languages) have been developed to deal with the problem of treating scattered data as a contiguous string, but they pay a very heavy price in memory overhead (in some schemes over three-fourths of the available memory is required to handle the addressing mechanisms) and in the processing time required to perform the address arithmetic.

An alternative solution is proposed here which involves the addition of a small Associative Memory (AM) to the addressing machinery of the computer (or peripheral direct access storage device). As will be shown, this hardware modification will permit scattered data to appear contiguous, with only a token overhead cost in memory and processing time.

We note that the data which are to be treated as a contiguous string can frequently be stored as a reasonably small number of physically contiguous strings. Further, the computer operation for moving through a physically contiguous string is the "indexing" operation. That is, a special register adds a fixed constant to the address of the current data element to obtain the address of the succeeding data element. Let us now assume that we wish to treat (individually) physically contiguous strings \( A_1, A_2, ..., A_N \) as a single string. We can do this by loading a map of the type shown in Figure 1 into an AM which monitors the contents of the effective address register of the computer's addressing mechanism. When an indexing operation results in a match with the search field of the AM, a microprogram interrupt occurs during which the tag field of the corresponding entry in the AM replaces the contents of the effective address register. The AM is searched in parallel, and essentially no time delay occurs in processing when there is no match.

Since the strings in this example were not specified, they can represent free storage as well as data. Thus, by simply changing the contents of the map loaded into the AM, the topology of the direct access storage device (be it CPU core memory or peripheral storage) can be altered to simplify and speed up the accessing, storage, and processing of virtual or actual data strings.

The VTRAM (Variable Topology Random Access Memory) concept presented in this paper will be most effective in those applications in which the number of physical and/or logical breaks in the contiguity of a data string is small compared to the number of elements in the string, and also does not often exceed the capacity of the supporting AM (Associate Memory). Data structures of this type, where breaks are the exception, occur very commonly.

Logical breaks can be handled by the VTRAM in the same way as physical breaks. However, for some situations we might desire that a logical break be conditional. This can be accomplished by appending an extra control bit to the break addresses stored in the VTRAM that permits address exchange for these

\[ * \text{See Appendix I for further exposition of this concept.} \]

\[ ** \text{A logical break in a string is defined here as either an interior entry point in the string, or a "jump" from one interior point in the string to another point in the string.} \]
Implications of a VTRAM for data manipulation

A generalized move operation (CPU to CPU core)

It is frequently necessary or desirable to move a logical data string, unaltered, from one physical location in CPU core to another. Data may be moved from one specific area of core into another prior to bringing in an overlay, or in order to convert a logical data string to a physical string prior to a channel operation. (A single transfer of data from CPU core to a secondary storage medium or output device frequently requires that the data be physically contiguous.) Data may also be moved to reduce the size of the map or the complexity of the address arithmetic necessary to visit the elements of a data string, or simply to reset to zero (or some other fill character) some portion of a data string. While the existence of a VTRAM can eliminate the need for many of these data moves, a significant residue will still be left.

A generalized move can be executed in a VTRAM by simultaneously loading the maps describing both the input and output strings into the AM. The number of bytes to be moved can be specified in the move instruction, or it can be determined by having the AM create an operation interrupt (rather than an address modification) when a match is detected between a specially flagged entry and the current core address. A more detailed treatment of this topic is given in a later section.

Insertions and concatenations from free storage

When storage requirements for a data string are determined dynamically, the area allocated to hold a given string can be exceeded, leading to the necessity of either moving the string to a new (larger) area, or setting up the machinery to handle data chaining. As noted in the introduction, software procedures for data chaining are expensive in storage and processing time overhead. In a VTRAM, free storage is made available to a data string by simply deleting a section of the map of free storage and concatenating it onto the map for the data string after making the initial and final address linkages. It is important to note here that the map representing a data string (in a VTRAM) does not have to be ordered. That is, the ordering of the entries in the map describing the string need have no correspondence with the logical sequencing of characters within the string.

The management of free storage is a recurring problem common to a wide variety of data processing systems. While the VTRAM concept does not significantly alter the nature of this problem, it does permit some simplifications in its resolution. Appendix II discusses this matter in greater detail.

Generalized data paths

It is not unusual to find problems in which the logical connectivity among a set of data elements is more complex than in the simple linear strings discussed previously. Consider the problem of processing a data ring whose elements are physically contiguous. A ring is a string whose last physical element is logically assumed to precede the first; further, the first and last physical data positions have no special logical significance.

In moving through such a structure in a conventional memory, a check must be made after each (address) index operation to see if a jump to the first physical element of the data set is required. In a VTRAM, a single entry in the AM will cause this jump to occur automatically when required.

Perhaps a more important consideration is the case where data stored physically in one configuration must be visited logically along a number of different paths in the course of processing. Without a VTRAM, a rather complex (and time consuming) series of instructions would be required to pick out, in turn, each of the desired paths. With a VTRAM, any given path can be specified by simply loading the map for that path into the AM. Some examples illustrating this discussion are given in the next section.
Channel operations and peripheral storage

The potential application of the VTRAM to the simplification of channel operations is especially significant. The necessity to physically group data (or use some form of channel data chaining, difficult to predict in a dynamic situation) is no longer necessary. The scatter storing of data can similarly be handled in a single operation by the hardware interpretation of the storage map.

To illustrate some of these concepts, let us consider a simplified example.

Assume we have a time-sharing system backed up by a bulk core memory logically partitioned into blocks, each of which is 0.2 the size of CPU core. Thus any load module (represented by a single map) can contain a maximum of five physical substrings. During a swap operation with CPU core, a scatter read (or write) from (or into) the peripheral store can be accomplished by a single channel operation using a five-word AM. A later section is concerned with VTRAM applications to peripheral storage.

Patching of “Slow-Write” peripheral storage

One of the potentially useful applications of the VTRAM concept is as an adjunct to optical peripheral mass storage devices. The relatively slow writing times make changing data undesirable, especially for making small corrections or insertions. The logical restructuring of data strings via the VTRAM would permit temporary “fixes.” For example, the fix could be stored in a small, fast auxiliary memory until enough alterations have been collected to warrant the generation of a revised “memory plate.”

VTRAM data processing

The following terminology will be adopted for the rest of this paper:

- A **string** is a linear sequence of elements, where linearity is a logical concept independent of the physical arrangement of the elements.
- A **block** is a string or substring which is physically contiguous, i.e., the logical ordering of the elements corresponds to their physical placement.
- A **map** is a string of break-transfer addresses which describes the organization of another string; one or more maps can be concurrently loaded in the AM to drive the address-exchanging mechanism of the VTRAM.
- A **file** is a string of maps, and is a unit for loading or storing the contents of the AM.

To present a concrete discussion, let us assume in this section that we are dealing with an IBM System/360 computer equipped with a VTRAM. The VTRAM has a small number of associative memory words (on the order of 10 to 50), each of which has enough bits to represent any valid core address. Corresponding to each word of associative memory, there is a tag memory word which can also hold an address (the “transfer” address) plus several bits of control information. One such bit is the “end of string” bit which signals to the VTRAM that this break address constitutes the end of a data string. A second is the “end of file” bit which denotes the end of a string of maps. (The VTRAM may contain several such files at one time, each file typically containing one or more maps, each map describing a string of data.) Finally, two or three bits can be used as a map key.

The VTRAM, of course, has the property that it is searched for a match in parallel, and essentially no overhead is incurred when no match is found. The overhead in replacing the content of the register being monitored (the “effective address register”) by the content of the tag field of the VTRAM is also negligible. The associative memory will only have to respond to an “equal” condition, thus reducing the cost of the unit. In addition, the associative memory will have the property that on being loaded it will use the first available nonzero registers; a property usually obtainable at no extra cost and which is very useful, as we shall observe.

Some additional IBM 360 instructions

The following commands have to be added to the CPU repertoire to manipulate the VTRAM:

1. **ON/OFF Associative Memory**
2. **Clear Associative Memory (CAM)**. This will replace the content of the associative memory registers by zeros. A selective clear based on the map key can also be requested.
3. **Load Associative Memory (LAM)**, from core location T. T is the starting location of a map. Note that this map automatically orders the storage and does not itself have to be ordered; the order of occurrence of the break-transfer addresses in the map is of no consequence. This is a very important property of the VTRAM, as it greatly facilitates additions and deletions to the map.

The loading of the associative memory continues until an end-of-file bit is encountered in one of the entries. If the available associative memory is exhausted before the end-of-file
indication is reached, an interrupt occurs (or a condition code set) and the address of the last entry loaded is posted. The possible remedies to this overflow condition will be discussed. Note that the VTRAM can be loaded with several maps corresponding to several logical data strings, at the same time. This facilitates storage-to-storage operations.

A highly useful variant on the LAM instruction is a BLAM (Backward Load Associative Memory), in which the break-transfer address pairs, as they appear in the table, are interchanged when loaded into the associative memory. This facilitates the backward scanning of a logically contiguous block or string of data.

4. Store Associative Memory (SAM), used by the supervisor during interrupts.

The augmented MVC command

To make use of all of this machinery, the storage-to-storage instructions (SS) of the 360 will be assumed to operate using the VTRAM. This means that the length specification will become unnecessary, since a string-end condition is now explicitly represented by break address entry with a flag.

A typical sequence of commands for, say, moving data from string A to string B (the strings possibly consisting of several blocks each) would be

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAM</td>
<td>Clear the associative memory (optional)</td>
</tr>
<tr>
<td>LAM TA</td>
<td>Load the map which describes string A</td>
</tr>
<tr>
<td>LAM TB</td>
<td>Load the map of string B (note that this is in addition to TA)</td>
</tr>
<tr>
<td>MVC B, A</td>
<td>Start move operation, with the initial addresses being A and B. (Note that the VTRAM contains only the break addresses, and not the initial addresses of the strings.)</td>
</tr>
</tbody>
</table>

The MVC instruction will terminate as soon as an end-of-string indicator is reached for either string A or string B, returning a condition code corresponding to each applicable case. An alternate possibility, in case the end-of-string for A is encountered first, is to fill the rest of B with, say, zeros.

Matrix multiplication

As an example of generalized data paths, we will look at a procedure for forming a product of two matrices, \( C = A \times B \). The three matrices are not assumed to be located in contiguous storage, but may rather be scattered all over core. We will however, assume that each row of the A and B matrices is in one block. (The matrices are \( n \times n \), with each element being a half-word integer, so that each row is \( 2n \)-bytes long.)

<table>
<thead>
<tr>
<th>Registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA, A</td>
<td>Register RA points to ( A(1, 1) )</td>
</tr>
<tr>
<td>RC, C</td>
<td>Register RC points to ( C(1, 1) )</td>
</tr>
<tr>
<td>R1, n</td>
<td>R1 is the counter for the I loop</td>
</tr>
<tr>
<td>R2, n</td>
<td>R2 is the counter for the J loop</td>
</tr>
<tr>
<td>R6, R6</td>
<td>R6 has displacement from start of row for ( B(K, J) ) for ( J = 1 )</td>
</tr>
<tr>
<td>R4, R4</td>
<td>R4 has the partial sum for ( C(I, J) = \sum A(I, K) \times B(K, J) )</td>
</tr>
<tr>
<td>RB, B</td>
<td>Register RB points to ( B(1, 1) )</td>
</tr>
<tr>
<td>R7, RA</td>
<td>R7 is the pointer to ( A(I, K) ) for ( K = 1 )</td>
</tr>
<tr>
<td>R3, n</td>
<td>R3 is the counter for the K loop</td>
</tr>
<tr>
<td>RB, 2n(RB)</td>
<td>Step row address. Note that if a break address is reached, the new transfer address gets inserted into RB from the effective address register. Thus, the next time through the loop, when the LA R5, 0(RB, R6) instruction is executed, the correct address will appear in R5</td>
</tr>
<tr>
<td>R7, 2(R7)</td>
<td>Move pointer to ( A(I, K) ) to next element in row</td>
</tr>
<tr>
<td>R3, LOOPK</td>
<td>Go through the k-loop n times</td>
</tr>
</tbody>
</table>
Because of the VTRAM device, this program did not differ substantially from any matrix multiplication program for contiguous stored matrices. Of course, similar procedures are used on computers not equipped with a VTRAM. These, however, require that the starting row address for each row of A, B, and C be stored in a table; with the VTRAM, only the actual break addresses need to be stored. An additional advantage obtained by the VTRAM is that the C matrix can be scatter-loaded randomly, without the restriction that no row can be broken.

Note that in the example it was necessary to return the break address modifications to one of the index registers used to control the addressing sequence because the data path was traversed in a sequence of instructions under program control, rather than as a single instruction triggering a micro instruction sequence. This was accomplished by means of the LOAD ADDRESS instruction which sums a fixed displacement and up to two index register values in the effective address register and then returns this sum (in our case, to one of the participating registers). When the effective address computed in this process matches a VTRAM break address, the effective address is replaced with the corresponding tag address, and this address, in turn, is transmitted back to the index register specified as the receiver for the LA instructions.

Another example with logical data path

As a second example involving generalized data paths, consider the problem of analyzing the contour of a two-dimensional graphical object, which is represented in storage by a series of fixed-length records, each giving the (x, y) coordinates of a point on the contour together with some additional information such as the identity of the line segment to which the point belongs. (The contour is partitioned into a series of line segments which are physically contiguous sub-strings.) The order in which the records are stored corresponds to the sequence in which the points occur on the contour of the object, with some arbitrary break point to permit storing this file structure, which is logically a ring, as a (physical) linear string. We assume here that there are no physical breaks other than the one which closes the ring.

In one operation we might prefer to start with the point having the greatest coordinate value, and count the number of points for which the coordinate monotonically decreases. Without a VTRAM, after each point is examined, we must check for the ring-closing physical break address in the table before indexing to the next point. With a VTRAM, storage of the break address would cause an automatic return from the end of the string back to the beginning if an attempt was made to index past this last entry in the file.

In a second operation, we might prefer to find the center of gravity of some selected subset of line segments. Without the VTRAM, we would be required to check the segment ID of each point against the list of desired segment numbers before indexing to the next point. With a VTRAM, storage of the break address would cause an automatic return from the end of the string back to the beginning if an attempt was made to index past this last entry in the file.

The overload problem

When a map for a given string gets larger than the available number of associative memory registers, it has to be partitioned so that the consecutive partitions of the map (submaps) now correspond to the logical ordering within the string. Given such an organization of the map, only a part of it need be loaded.
into the VTRAM at one time; if a given operation is not completed before the last break address corresponding to the end of a submap is reached, the next submap replaces the old one, and the operation is repeated. This process continues until the end-of-map indicator is encountered.

When a large map is thus partitioned into submaps, each of which is small enough to fit into the VTRAM, it should be noted that there is no need for the submaps to be ordered internally. This fact facilitates the sorting process, and also allows subsequent insertions of new break-transfer entries at any available position within the appropriate submap.

Note, however, that even the partial ordering of a map entails what is usually a very large overhead, and the VTRAM should contain enough registers to obviate the necessity for this ordering in the great majority of cases. If it were not for this fact, we could always assume that the maps are ordered, and would be able to operate using an associative memory with only one register; i.e., any ordinary register which can monitor the effective address register would suit our purpose.*

The reordering operation may itself make use of the available associative memory. The use of associative memories to facilitate sort operations has been studied in the literature.1

Some limitations

While the VTRAM mechanisms presented here are well suited to the separate handling of logical or physical breaks, the combination of such breaks in the same string will cause special problems. To specify a direct jump from the jth element of a string to the jth element across a physical break, some address arithmetic using the string storage map must be performed. The arithmetic is simple and needs to be performed only once. Nevertheless, the computation is time consuming and sometimes cannot be done at the time the string is stored, but must be made when the jump is required. For these reasons, an executable instruction string which contains many dynamically determined branches is not a desirable candidate for VTRAM-controlled scatter loading.

Using a VTRAM for peripheral storage management

A key problem which arises in many different contexts in dealing with random access storage devices is the fact that there is really no such thing as a truly random access device; all existing devices are actually organized into groups of records, each one of which is bounded on either side by other records. This causes problems when the information contained in a record changes dynamically in size; for example, it may no longer fit into its former place. One is now typically faced with a decision of whether to invest the time in trying to make it fit into one place (either by pushing aside its neighbors or by finding a different place for it which is large enough), or to string it out in several records by chaining the various pieces, thereby sacrificing time during retrieval. A VTRAM can be used here to great advantage to create logical linkages between physical records which are not physically contiguous, thus allowing them to be written or read using but one CPU channel command; i.e., to make them look like a single physical record to the computer.

We will discuss two classes of random-access storage devices; a bulk core memory, and a fixed-head rotating memory. In both cases we shall assume that the CPU communicates with a controller unit for the memory in question, and that this controller is in reality itself a small stored-program computer; it is this small computer that we wish to equip with a VTRAM in order to facilitate the restructuring of records without actually moving data around. It should be pointed out that the controller will be able to perform a certain amount of housekeeping operations "for free" if it can do them subsequent to the completion of a given I/O operation, since there will be periods when the CPU is not using this class of devices. If the controller can schedule housekeeping operations, for example by waiting a certain time interval to give the CPU a chance to start a second operation which might have been present in a queue, more efficiency can be achieved. Thus, we can in general tolerate more overhead for housekeeping in the controller than in the CPU. On the other hand, for a rotating device, the mechanical aspects of the memory sometimes present critical timing problems for the controller.

Drum memory

To lend concreteness to our discussion, we shall describe a specific rotating memory. This is a drum, with 800 tracks, each of which is somewhat over 4096 bytes long (the exact length will depend on the particular memory organization that we choose). The drum is rotating along the tracks, so that a given track can be read during a single revolution. Only one head can be reading or writing at any given time, but heads can be switched at any time at electronic speeds; hence,
during a single rotation data can be picked up from several tracks, but necessarily from different positions along the tracks.

We shall assume that the drum is equipped with a timing track, whose current content together with the current track indicator are being read into a register which can be monitored by a VTRAM. (This register will serve the same function as the “effective address register” already discussed.)

Along each track, memory will be divided into sectors. A sector is a quantum of storage, say 64 bytes long; thus our drum has 64 sectors per track. Since, by definition, a sector is the smallest addressable unit of storage the timing track need only carry information as to the current (or next) sector.

Because of the possible time lost in head switching, there may actually have to be a physical separation (amounting perhaps to several byte positions) between the end of a sector and the start of the next one.

The first part of each record consists of its storage map. (See Figure 2.) The record is addressed (by the computer) by giving the address of the sector which contains the map. The map is read into the VTRAM in the controller; it then controls the mechanics for reading the rest of the record by causing head switching to take place. If the map terminates in the middle of a sector, the data portion of the record starts immediately. If only retrieval is desired, the data transmission to the computer need not commence until after the “end-of-map” indicator has been read into the controller. For a write operation, the computer will have to supply not only the address of the start of the record, but the map as well. The onus of storage management on the drum, and the decisions as to where to put the various pieces of a record, must be left up to the central processor. The controller is capable of performing this function equally well, but the efficiency of use of a rotating memory depends critically upon the scheduling of accesses to the memory. To do intelligent scheduling, the controller would have to have access to a much larger body of dynamically changing information than is normally feasible (except on computers like the CDC 6600, where the controller is itself a full-fledged CPU with full access to all of core).

Let us look at the case of a record which is being rewritten, but which requires more space than it formerly occupied. Instead of releasing the record’s current storage to the free storage pool and then allocating a contiguous block of storage for the entire record, as might be done on devices without a VTRAM, the drum storage allocator of the CPU would be asked to allocate additional storage from a longitudinal position which begins immediately following the termination of the last sector in the current record. (Various alternate strategies are possible if such storage is not currently available.) That is, a piece is simply added on at the end of the existing record, thus making a longer record. Note that this piece can, in our case, come from a selection of any one of 800 different sectors.

The allocation procedure has the very important property that the new storage block is always added on at the logical end of the current storage. What this means is that our maps are always ordered—successive entries in the map represent successive blocks of the record. Thus, there is no need to use the entire break-address portion of the map to monitor the timing track. Instead, the controller can have a pushdown stack, in which only the top element of the stack is used to monitor the effective address register. When a break address is encountered, the next element (break-transfer address pair) is popped up and becomes the new top of stack.

A basic underlying assumption in this discussion is that the entire record is rewritten during an output operation. This distinguishes the use of the VTRAM device in the present case from its use for core-to-core operations as discussed in a previous section, where a logical unit of data could be restructured dynamically in many ways other than adding on at the end.

To recapitulate our proposed method for handling drum storage: each record carries along (at its beginning) a map of where the record is located; this map is loaded into the VTRAM device and is used for automatic head switching as required. There is essentially no overhead lost in retrieval time in this scatter storage of a record compared to having the entire record stored on a single track. There is, however, some overhead paid in storage. For our drum, each map entry would
is required for each sector, the storage overhead is somewhat under five percent.

**Bulk core memory as peripheral storage**

This type of storage device is characterized by the fact that there is no latency time due to rotational delay to worry about, as was the case for the drum. Thus, as far as the computer is concerned, all storage locations are equally accessible, and it does not matter where various pieces of data are stored. As in the case of the drum, however, we will assume that the computer deals with storage in records, and that a single record is to be fetched or written with a single channel command. It is, of course, possible to assign much more complex structure to the data, and have this structure reflected in the nature of the computer-controller communications. As this would unnecessarily complicate the ensuing discussion without contributing anything, we will assume the simplest possible structure in the bulk core.

Using a VTRAM enables us to have a record consist of many noncontiguous blocks. For bulk core, all free storage management can be left up to the controller. We propose a similar record format as for the drum, where the computer addresses a “header” which contains a map of the record; the computer, however, need never see the map. The free storage handling strategies discussed in Appendix II for CPU core are equally valid for the bulk core as well. The significant differences for bulk core are (1) more time can be invested in periodic recondense operations since they can be scheduled “off-line” by the controller, and (2) the VTRAM implementation need not actually involve any associative memory, but only a pushdown stack.

When the CPU initiates a bulk core write operation, the free-storage-map (held permanently in a pushdown stack) is used by the VTRAM to direct the store operation. The top of the free-storage-map, which corresponds to the storage blocks needed to contain the transmitted record, now becomes the map for this record.

In the case of the bulk core, the most convenient location for the map describing a record is immediately following the record. However, since the map itself is a record which may have to be scatter loaded, the initial segment of any such map must contain the break addresses needed to retrieve the map. (This initial segment terminates with a special flag bit.) Thus, after a write operation has been completed, the controller adds to the top of the map for the stored record a prefix (possibly null) of additional break addresses and this augmented map is stored (under control of the VTRAM and the map prefix) immediately following the record it describes. Finally, the address of the first entry of the prefix is returned to the CPU as the address to be used in reading the record. This address is available for returning to the CPU immediately at the conclusion of the write operation, even though the contents of this location will not be determined until after the map itself has been written.

For a read operation, the CPU-supplied address is used to obtain the map prefix which then directs the loading of the map itself into a VTRAM pushdown stack. Now the map directs the requested read operation.

**Comparison with other address-mapping schemes**

Associative registers and other special-purpose memory addressing hardware are currently employed in a number of computing systems (e.g., IBM 360/67, GE 645, B 8500) to implement address mapping for paging and segmentation schemes. These concepts and their associated hardware organizations are significantly different from the VTRAM concept presented here. Paging is a scheme for making the fast core appear larger than it actually is (virtual memory), and is accomplished by defining a mapping function from a large virtual space into the small physical space. The paging hardware is employed at every memory reference and at least one conversion per reference is required to translate the symbolic address contained in the instruction into a physical address. Pages of core are usually of a fixed size, although they may come in several sizes. This organization allows jumps into the middle of a page to be handled easily. Maps (segment and page tables) pose a special handling problem and entail storage overheads.

The VTRAM concept is concerned with dynamically restructuring the logical contiguity relations between data within the same real space without having to move the data. A second application of the VTRAM concept is to reduce the programming and timing requirements for moving physically scattered data between a peripheral device and core, or from one set of core locations to another. The VTRAM hardware is used to exchange one physical address for another only at the break points specified by the active storage maps; otherwise it does not intervene in the addressing process. The length of each block of storage is of no consequence. The VTRAM is intended to support operations which index through the data, and unexpected logical breaks can be handled only with difficulty. The storage maps
used by the VTRAM can themselves be scatter loaded just like any other data string; this permits a saving in space, for even though the maps may dynamically change in size, no contiguous blocks of storage need be reserved for them.

CONCLUSIONS

The VTRAM concept, as developed in this paper, is a hardware organization for achieving, on almost any digital data processor, a string processing capability, extending to channel and peripheral operations, at very low cost. The central idea is that of "address exchange" when a critical boundary is crossed during an indexing operation. Implementation is achieved by storing these boundaries in a small associative memory so that many of these boundaries can be searched for in parallel, thus avoiding any significant processing time overhead. Given the presence of the AM, it is also available for use in a more conventional manner.

The major advantage gained via the VTRAM is the ability to do "string processing by exception" for strings which contain relatively few breaks.

Such strings (the authors feel) are very common, and come about as a result of insertion, deletion, or rearrangement operations on formerly contiguous data. They also arise when contiguous data, stored physically in one configuration, must be visited logically along one or more different paths in the course of processing. There are types of strings, however, such as those consisting of executable code, for which the VTRAM mechanization as described here is not very useful.

One area where the authors feel the VTRAM concept has much potential value is in augmenting data transfer machinery.

It is impossible to give a simple answer to the question of how much core is required to adequately support a given CPU, and similarly the question of how much AM is required to optimally implement the VTRAM concept with a given amount of core cannot be answered except in very specific situations. The authors feel that there are many significant applications where a small fixed AM, say 10 to 50 words, would be very valuable regardless of the amount of core.

For the VTRAM concept to be useful, the AM size must be large enough to contain the maps generated by the various applications. This will be ensured for most cases by free storage management procedures and by programming limitations dictated by the actual size of the AM. In the unusual case of severe fragmentation of a given storage area, causing the AM size to be exceeded, several procedures utilizing the VTRAM are suggested.

An especially important advantage of the VTRAM concept is its applicability to almost any reasonably sized computing system with very little hardware modification.

ACKNOWLEDGMENTS

The authors wish to express their gratitude to Professor Harold Stone of Stanford University, and to others who read the preliminary draft, for some very cogent comments and ideas which contributed to this final paper. Thanks are also due to Margarett N. Collins for her able editing of the manuscript.

REFERENCES

1 R R SEEBER A B LINDQUIST

Associative memory with ordered retrieval

2 P DENNING

The effects of scheduling on file memory operations
Proc S J C C 1967

3 E L GLAZER J F COULEUR G A OLIVER

System design of a computer for time-sharing applications

4 J D MCCULLOUGH H K SPEIERMAN

F W ZURCHER

A design for a multiple user multiprocessor system
Proc F J C C Part I Vol 30 1965 611-617

5 W T COMFORT

A computing system design for user service
Proc F J C C Part I Vol 30 1965 619-626

6 D E KNUTH

The art of computer programming, vol 1: Fundamental Algorithms
Addison-Wesley 1968

APPENDIX I: VTRAM IMPLEMENTATION

Figure 3 shows a typical computer data flow organization augmented by an AM. The main point to note here is that implementation of the VTRAM concept can be accomplished with minimal disturbance to the conventional data flow paths. The essential requirements are a connection to the main memory output bus to permit the loading of the AM, and a connection to the Address Adder (or "effective address register") so that monitoring and address modification can be accomplished.

To more fully utilize the capabilities of the AM, it may be desirable to introduce additional direct paths to one or more of the index registers. However, all of the operations discussed in this paper can be carried out without these additional connections.
APPENDIX II: STORAGE MANAGEMENT STRATEGIES USING A VTRAM

A major function of the VTRAM device is to facilitate the dynamic restructuring of data, without actually having to move the data about from place to place. A very important application of this concept is in storage management for complex dynamic systems, such as a time-shared executive system. A rough description of the environment might be as follows. (This is a cross section at time t0.)

- There is a collection of storage called “free storage” which is noncontiguous.
- There are n jobs, each of which occupies some noncontiguous region of storage. These regions are mutually disjoint, and do not intersect free storage.
- Each job may request additional storage from free storage. The request will be for a certain amount of storage, and may be allocated (by the executive) at its discretion from anywhere in free storage. When such an allocation is made, the storage area is deleted from free storage and becomes part of the job’s storage region.
- Each job may release any or all of its storage to free storage.

It should be pointed out again that a VTRAM is not particularly useful for executing scatter-loaded code, as program strings tend to have too many logical breaks (interior entry points). The reader should assume for this discussion that the jobs are requesting core for data manipulation. For example, a job may wish to read in a record from a peripheral storage device, insert a new field, and write the record back into the storage device. Note that this operation is precisely of the form discussed above: only one or two breaks are introduced into a (formerly) contiguous data string.

Let us discuss the free storage management function. This management must perform two distinct functions: that of allocation and that of releasing storage back to free storage.

The management of free storage is concerned with creating a balance between the tendency for blocks of storage to become progressively smaller (through the randomizing action of the allocation-release process) and the overhead involved in rebuilding larger blocks from the available fragments.

The overhead for condensing free storage can be paid in a number of different ways, involving such considerations as the time required to allocate or release storage, the frequency with which the condensation must be repeated to keep the average block size above some minimal value, and the complexity of the hardware and algorithmic procedures needed to perform the condensing operation.

Assuming, as we are doing, that each job will manipulate its own data structures using the VTRAM, which has a limited storage capacity, it is incumbent upon the executive to minimize the storage fragmentation during the allocation function; otherwise, each job will have its storage broken into so many noncontiguous pieces that the storage capacity of the VTRAM will be exceeded very often, with a resulting high overhead, thus negating the benefits from having this device available. On the other hand, both the allocation and the releasing function are performed so frequently that if the system is to function efficiently, these functions must not take too much time.

The strategy discussed below is derived by adding a VTRAM to the commonly used “first fit” strategy: e.g., see KNUUTH, Section 2.5.

Free storage is represented as one string (noncontiguous), with a map of break-transfer addresses. When a request for n storage units comes in, storage is “peeled off” from the top by assigning as many blocks as are required to satisfy the request. Since the end of the request will typically fall in the middle of a block, a new break-address will terminate the block given to the job, and a new start address will be assigned for free storage. This allocation operation is extremely fast.

The storage release operation itself is equally simple. The job denotes the area it wished released by supplying a map; this map is appended to the free storage map...
and the operation is finished. (The reader may note that this is conceptually identical to a "threaded list" organization of free storage.) Since the area required for the free storage map itself necessarily has to be finite, the VTRAM can be used to handle this area in a cyclical fashion (with a fixed maximum size for the number of entries). Thus, as entries are taken off the top and added at the bottom, the two operations are eventually performed at the same rate, and a steady-state cyclical storage area suffices.

The disadvantage of this strategy is that storage will become increasingly more fragmented, since no attempt is made to find the best blocks for this particular request. If this approach is to be made workable, a "garbage collection" operation must be performed periodically after storage is released.

To facilitate the garbage collection, it is useful to represent a storage map as an (unordered) collection of ordered triplets (B_i,T_i,AL_i), where B_i and T_i are the break-transfer addresses as before, and AL_i is the address of the last element of the block started by T_i. That is, every AL_i is equal to some B_j in a one-to-one fashion. In addition, the start of a block is represented by a triplet (0,T_0,AL_0), where the break address is empty. The reason for this redundancy is that the break-transfer addresses do not lend themselves to a convenient identification between a T_i which starts a given piece and a B_j which terminates it. Of course, during the use of the VTRAM, only the B_i's are loaded into the associative memory, and only the T_i's need appear in the tag part.

To return to the garbage collection algorithm: The operation of combining contiguous blocks consists simply of looking for matches between the AL_i's and the T_i's. Anytime that a match T_j = AL_i occurs, we simply replace AL_i by AL_j and delete the triplet (B_j, T_j,AL_j) from the map. Most of the overhead in this operation is in the search operation; a function greatly facilitated by the presence of the associative memory.

After the garbage collection, free storage has been completely condensed and consists of a number of noncontiguous areas, each of which is represented by an entry in the map. Note, however, that the ordering of the blocks is totally random, since there is no advantage to be gained in sorting these blocks on their respective core addresses; nor is there anything to be gained in sorting them by size, since the allocation strategy calls for assigning storage from the top of the free storage list.

If the associative memory is large enough to accommodate the entire T_i vector, the loading operation during the search for matches needs to done only once, thus speeding up the condensing process considerably. It might therefore pay to do the condensing quite frequently, in order to keep down the size of the T_i vector (as well as to cut down on the fragmentation of the allocated storage). The exact tradeoffs involved (i.e., what is a reasonable size for the associative memory, and how frequently should one recondense, given certain assumptions about the job mix and probability distributions for storage request-release operations) will be the subject of a simulation study conducted by the authors, with the results described in a forthcoming paper.

In summary: this strategy for storage allocation and the corresponding representation of storage blocks is characterized by extremely rapid handling of requests and releases. The storage representation is in a form directly utilizable by the jobs because of the presence of the VTRAM device. Some overhead is incurred for the periodic recondense operation associated with storage release, but this overhead is considerably reduced by the presence of the associative memory.

The overall efficiency of this method will depend upon the size of the associative memory; the exact relationship is unknown pending the outcome of a simulation study.