STORAGE REALLOCATION IN HIERARCHICAL ASSOCIATIVE MEMORIES

Jeffrey L. Gertz
Bell Telephone Laboratories, Incorporated
Holmdel, New Jersey

Summary

Two recent trends in computing, namely parallelism and programming generality, imply that increasing importance will be placed on developing location-independent schemes for computer memories. This paper examines several issues that arise when hierarchical associative memories are used to accomplish this objective. First, it presents methods for constructing economical associative memories to be utilized as lower levels in the hierarchy. Then, it considers the ramifications of storage reallocation in the memory and develops a new unit of storage transfer, the paragraph. Finally, it demonstrates that difficulties that might arise due to duplicate storage of data words in the memory can be negated by a simple scheme.

Introduction

Recent work by Dennis[1] has indicated that two current trends in computing require a radical change in thinking in regard to computer system architecture. These are the increasing importance of parallelism in computer operations and the growing desire for programming generality in computer system usage. Both trends imply that increasingly greater importance will be placed on developing location-independent schemes for computer memories. One possible manner of accomplishing this objective is to employ an associative memory for the computer system.

Parallelism in computer systems is desirable for several reasons. Some of the more important of these are: hardware utilization is increased because resources can be shared, information can be shared among several computations at once thereby reducing memory requirements, and individual programs can be run more rapidly by executing instructions in parallel whenever possible. Although parallelism can operate within standard addressable memories, much efficiency is sacrificed by such implementations. In particular, addressing restricts allocation choices, yet dynamic storage allocation is vital for multiprogrammed systems, and memory and processing efficiency are lowered by the many mapping tables required by addressable memory systems.

A program that exhibits programming generality should be able to be run on any sufficiently powerful computer and be usable as a building block for more complex programs without its internal structure being known. The latter property requires that a computer system allow a program module to create and transmit arbitrarily complex information structures as parameters to procedures whose storage requirements are unknown. These requirements imply that location-independent addressing must be used, since only the operating system of the computer could possibly make the necessary storage allocation decisions.

This paper is concerned with the examination of several issues that arise when the implementation of hierarchical associative memories is considered. First, it presents an overview of the manner in which the various levels of the memory might be constructed in an economical manner. Then, it considers the ramifications of storage reallocation in an associative memory, and in particular, develops a new unit of storage transfer, the paragraph. Finally, it investigates an issue that could be a serious problem in associative memory hierarchies, the duplicate storage of data words, and demonstrates that a simple scheme can negate any difficulties that might arise. A complete study of all issues involved in the analysis and design of hierarchical associative memories is contained in a previous report.[2]

Associative Memory Hierarchies

In recent years, as various feasibility studies have indicated the value of associative memories, several hardware associative memory implementations have been built or proposed.[3, 4] Three main areas in which investigation is currently proceeding are cryogenics, thin films, and semiconductors, although as many as a dozen other technologies are also being considered. Whatever the material actually used, all associative memories presently have two serious drawbacks: they are quite small by addressable memory standards, and they are very expensive. The first problem can be alleviated by using modularization, but the only solution to both is to use conventional addressable memories, suitably modified to appear associative, in several levels of the memory hierarchy. There are two main courses one can pursue in designing nonhardware associative memories for the lower levels of the memory hierarchy. One is to build a discrete, parallel search memory from relatively cheap components, such as shift registers. The other is to devise a method to use a standard addressable memory, such as a drum, as an associative memory. This section will develop each of these alternatives. For ease of discussion, the concrete examples just mentioned will be used in these formulations.

The simplest envisioned parallel search shift register associative memory, illustrated in Figure 1, can be constructed as follows. Each word of the memory

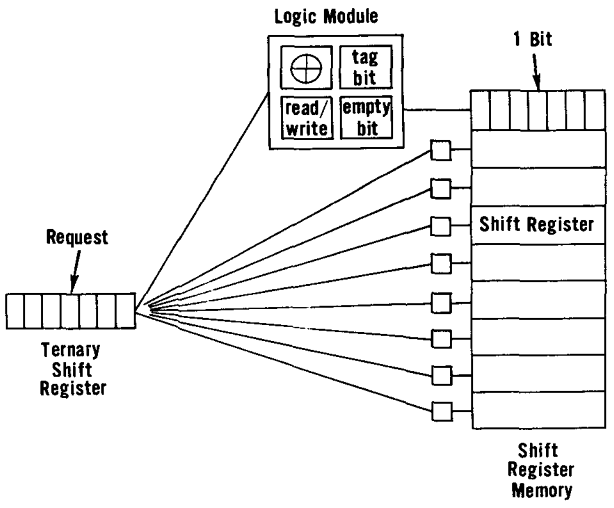


Figure 1 - Shift Register Associative Memory

will be stored in one shift register. A separate ternary shift register (0, 1, don't-care) will be used to hold the key to be matched. Associated with each memory word will be a logic module containing the following items: (1) an exclusive-or circuit; (2) a tag bit; (3) a word-empty bit; and (4) a read-write circuit. The actual memory operation would then consist of alternations of the following two cycles:

Cycle 1 - matching

(a) set to 1 the tag bit of each memory word whose word-empty bit is 0.

(b) compare the first bit of each memory word with the first element of the key.

(c) reset to 0 the tag bit of each memory word which doesn't match the key.

(d) shift all memory words and the key one position each.

(e) if n shifts have occurred, proceed to Cycle 2, else return to step (b).

Cycle 2 - reading

(a) read out the first bit of every memory word whose tag bit is 1; write an element of the next key into the key shift register.

(b) shift all memory words and the key one position each.

(c) if n shifts have not yet occurred, return to step (a).

(d) if readout is to be destructive, set to 1 the word-empty bit of every word whose tag bit is 1.

(e) proceed to Cycle 1.

If the key is to be read into memory instead of matched, then Cycle 1 is used to read the key into the first word whose word-empty bit is 1 while Cycle 2 is used only to place the next key in the key shift register.

The most efficient method that can be devised for the conversion of a drum memory from addressable to associative access requires a functional reversal of the usual associative memory operation. Namely, the words on the drum become the keys while the proffered keys become the associative memory. Basically, the envisioned system would work as follows. When a retrieval request is made to the memory, the key is placed in a request list which is implemented by an associative memory. Then, as each memory word is brought under a set of reading heads, it is treated as a key and associatively compared against the request list. If a match is recorded, the memory word is transmitted to the desired piece of hardware, most likely a memory unit in a higher level of the hierarchy. Of course, if the request list were full, a new request would be temporarily queued.

Writing in such a system could be implemented in many ways. Possibly the simplest of these is to keep the addresses of all empty locations in an empty list. Then, when a write request were made, the word to be written would be given an address from this list and the drum system would perform the write operation in just the same way it presently does for addressable operation. Correspondingly, when a destructive readout occurred, the newly available address would be

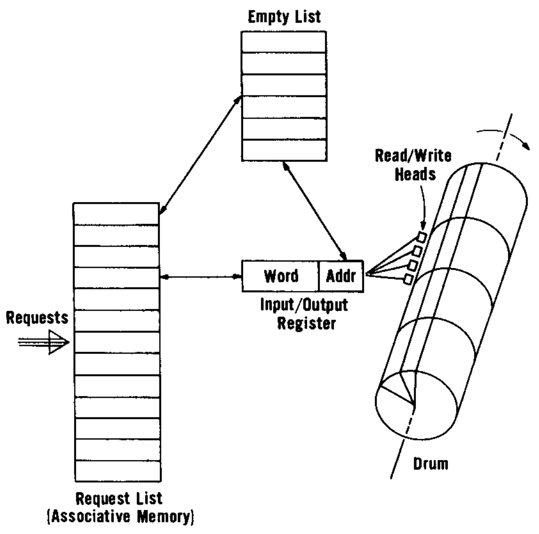


Figure 2 - Associative Drum Memory System

appended to the empty list. Finally, to revise a stored word, the word would first be read out destructively, changed, and then written in any available location, not necessarily the one from which it was read. Figure 2 gives an overview of this associative drum system.

Storage Reallocation Concepts

When a word not in first-level memory is sought by a processing unit, it must be brought up to that level by the memory hardware. In order to attempt to minimize the number of such time-consuming storage reallocations, this word is generally brought up with a suitably selected block of words. The selection scheme chosen should attempt to meet the following two (possibly divergent) objectives: (1) minimization of the time used for information transfer, and (2) minimization of the number of words elevated but not later used.

At present, the commonly used unit of information transfer is the page, a fixed-size block of contiguous words. However, the page has several serious drawbacks for our system. For example, when a program is represented as an information structure, the actual storage order of the instructions cannot be made to reflect their order of execution as the latter is not determined prior to runtime. Much information about the future needs of a procedure becomes available during runtime, but a page, being fixed in nature, cannot incorporate any of this knowledge. Also, the use of pointers as input parameters for a procedure, required for programming generality, implies that the procedure's data set may well be scattered throughout memory. However, the most serious drawback is contingent upon our use of an associative, rather than an addressable, memory. For an associative memory, the concept of contiguity is meaningless since words can only be accessed by content, not by relative location. To introduce addresses to an associative memory would only bring back all the disadvantages we sought to eliminate. Thus, the ideal unit of information transfer for our system should be based on content rather than location, be formed dynamically, and be able to incorporate words that may be physically separated in memory. To suggest these properties, we have coined the term paragraph.

Considerable, though not total, freedom exists in choosing words that belong to the same paragraph. The only restriction is due to a singular property of associative memories, namely that several words may be retrieved at once, as multiple matches to a key are possible (whereas addresses, of course, are unique). This could occur, for example, when accessing all data having a given property, or when retrieving all instructions logically following a newly completed one in a parallel processing environment. Thus, memory requests are satisfied by phrases, where a phrase is the set of all memory words which may be accessed by one key during the execution of some process. In addition, any word may be accessed by several keys, as a key may be to any subset of the fields contained in a word. Hence, a word may belong to several different phrases.

Clearly, all memory words in the same phrase must, of necessity, belong to the same paragraph. This, in turn, leads to a structural condition for a paragraph; namely, that it must consist of an integral number of phrases. This result can lead to serious trouble if no restrictions are put on the number of phrases to which any one word may belong, as a paragraph must contain every phrase of which any of its constituent words is a member. Thus, to prevent paragraphs from becoming enormous in size, and to prevent local changes from having global repercussions, we require the following postulate.

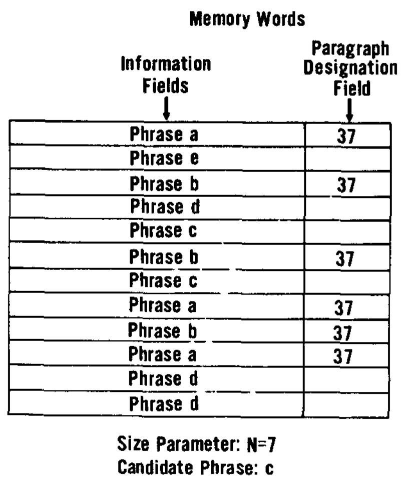
Postulate 1: No memory word can belong to more than one phrase whose size is greater than one.

Aside from the previous structural condition, no other constraint exists concerning which words may be grouped together in a paragraph except that the words be chosen in such a way that the previously detailed objectives be met in an optimal manner. Two distinct approaches appear worthy of pursuit. The first is the use of preset paragraphs which resemble pages in that they are formed prior to execution and have an approximately fixed size. The second is the use of dynamic paragraphs which fit more closely with our ideal unit in that they are formed when required and will vary during execution.

A preset paragraph will be defined as a block of N or fewer memory words that are moved together for storage reallocation purposes. It is constructed by grouping together as many phrases as possible that, based upon expected program behavior, are likely to be used together in time. By its nature, a preset paragraph maintains two of the important attributes of pages, namely an approximately fixed size and uniqueness (every memory word will be part of one and only one preset paragraph). In addition, it has two advantages over the page: contiguity of storage locations is not a binding factor and word additions may be more easily accommodated. Once a preset paragraph has been formed, a method is required that will allow the computer system to locate all of its words. The only feasible one is to append a paragraph designation field to each memory word. Then an entire paragraph can be retrieved with one memory access by keying on the desired designation field. Figure 3 depicts a sample preset paragraph stored in memory.

One of the disadvantages of using preset blocks of words for storage reallocation is that many words that are known at retrieval time not to be needed by any active computation in the near future are brought into first-level memory, thereby tying up valuable storage locations. The purpose of using dynamic paragraphs is to attempt to eliminate such inefficiencies of use of first-level storage. In an attempt to meet this objective, such paragraphs can be built in a "forward" direction from an accessed phrase. That is, when a phrase not in first-level memory is to be retrieved from lower memory, a paragraph is formed consisting of that phrase and the group of phrases that are most likely to be accessed in succession. This group of phrases will be referred to as the successor phrase set of the given phrase. Thus, a dynamic paragraph will be defined as a block of memory words constructed by adding the successor phrase sets of each phrase already included in the paragraph until some paragraph completion rule is met. With such construction the paragraph consists of a tree-like structure, as illustrated in Figure 4.

A major disadvantage of the dynamic paragraph is its lack of uniqueness. That is, a memory word may belong to several different paragraphs during its life time, a situation which would arise whenever the phras tree-structures of two different accessed phrases intersect. Should a word belong to more than one


Memory Words

Information Fields	Paragraph Designation Field
Phrase a	37
Phrase e	
Phrase b	37
Phrase d	
Phrase c	
Phrase b	37
Phrase c	
Phrase a	37
Phrase b	37
Phrase a	37
Phrase d	
Phrase d	

Size Parameter: N=7
Candidate Phrase: c

Figure 3 - Structure of a Typical Preset Paragraph

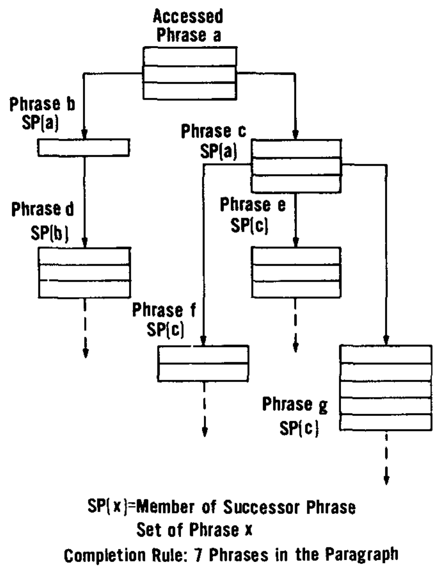


Figure 4 - Tree Structure of a Typical Dynamic Paragraph

paragraph at the same time, duplication of first-level storage would result. Thus, the following storage tradeoff should be considered when choosing the type of paragraph to employ -- a preset paragraph system wastes valuable first-level memory by storing words that are known to be unneeded while a dynamic paragraph system wastes memory by storing words more than once. Another disadvantage of the dynamic paragraph is that several accesses are needed to retrieve its words, one per each successor set in the paragraph. No designation scheme is required for these paragraphs, however, as their constituent phrases need not be removed together. Thus, a dynamic paragraph exists for only one instant of time, namely when it is formed.

Since neither type of paragraph has a fixed size, storage reallocation based on their use is more complex than that for pages. In particular, a means must exist for determining at any time the number of available storage locations in the main memory. This is because it will no longer be true that removing a paragraph from first-level memory will either be necessary or sufficient to provide room for a newly elevated one; small paragraphs may fit in the available space without displacing any paragraph, while larger ones may displace several small ones. Also, it may be necessary to count the number of words in the newly elevated paragraph. This can be avoided for preset paragraphs, however, by the use of the following algorithm.

1. If there are N or more available locations in memory, the newly elevated paragraph is stored immediately and no words are removed at this time.

2. Otherwise, the paragraph that has been designated as the first to be displaced is removed by the hardware.

3. Steps 1 and 2 are followed alternately until the new paragraph has been stored.

Thus, word elevation and removal can be performed at the same time using this method. Dynamic paragraphs can only utilize this simplification if the completion rule is based on size. Otherwise, word-counting must be employed. The removal algorithm for dynamic paragraphs can be made more efficient than that for preset paragraphs, however, as the removal can be made by individual phrase rather than by whole paragraph.

Multiple Storage Considerations

Many present day addressable computer systems do not actually "move" a page into main memory; rather, they create a duplicate copy of the page when it is required. The advantage of such a multiple storage system is that much reverse traffic in the interlevel communication links is eliminated, as unmodified pages (whose change indicators have not been set), particularly pure procedure pages, may simply be overwritten. The possible existence of several conflicting copies of a data word will never cause trouble, as the addressing mechanism of the computer system will always direct any access to the most recent version of the word.

The multiple storage strategy will result in the same increased operating efficiency if it is employed in associative memory systems. However, whereas this scheme is optional for addressable systems, it is required for associative ones which allow sharing of information. Otherwise, if a phrase existed in only one place in active memory (those levels accessible to the processing hardware), the execution time of an instruction could become unbounded. This unacceptable behavior could result from the following sequence of events:

1. Process A requests phrase X.

2. A memory search locates X in second-level memory.

3. Process B requests phrase X.

4. The new search fails to locate X in main memory.

5. Phrase X is paragraphed into main memory due to the search initiated by process A.

6. The search corresponding to B's request continues in the second level, with no success.

7. This search continues until all active memory has been checked, with no success.

At this point the system cannot decide whether phrase X is not in active memory or has been missed. Furthermore, if the search returns to first-level memory to check for the latter occurrence, the entire cycle could repeat once again.

This problem is resolved in addressable memory systems by having the supervisor maintain updated address tables. Then, if access to a specified lower-level location is unsuccessful, the search mechanism will know that a paging operation is in progress. No foolproof method appears to exist for preventing these occurrences in single storage associative memory systems. For instance, assume a list of all currently sought phrases were created. This would fail because phrases are often retrieved not by name but rather by their paragraph affiliation with explicitly sought phrases. Such implicitly sought phrases would of course be unknown to the system. When the multiple storage strategy is employed, a search that missed a phrase being paragraphed into main memory would locate the lower level copy, which it could then retrieve for the process initiating the search.

Unfortunately, the use of multiple storage creates a new problem, namely that two copies of a phrase may exist in first-level memory at the same time. This raises the possibility of the two copies becoming different, and thus of an instruction receiving an incorrect operand. This problem can be eliminated either by devising a scheme to prevent multiple first-level storage or by insuring that the multiple copies always exist in the same module (proved below). No foolproof method exists for implementing the former alternative, however, as the timing of actions cannot be controlled carefully enough to guarantee correct behavior (unless synchronous operation is used, an impossible approach for decentralized, modular computers). For example, the best scheme would appear to be to mark the left-behind copy of a phrase when it is elevated into main memory. Then future accesses that missed a phrase being paragraphed into first-level memory would note the mark and return its searching to that level. This method, of course, partly negates the main advantage of multiple storage, as unchanged paragraphs can no longer be simply overwritten since the mark on each component phrase must be removed from the lower level copy. More important, though, the search mechanism can continually oscillate between memory levels with this scheme if the timing between searching and paragraphing becomes out of phase.

Thus, since duplication of first-level storage cannot be satisfactorily eliminated, a method of operation that guarantees that multiple copies of a phrase always exist in the same module is required. Various set associative properties of phrases could be used to implement this policy, but only that of ownership meets all the conditions placed on modular associative memories. Every phrase must have an owner, as otherwise neither privacy nor security could be maintained in the computer system.

The reason why duplicate storage within a first-level module cannot cause system irregularities will now be explained. To do this, the following definitions are needed:

Definition 1: A computer system is conflict-free if every pair of instructions in the system satisfies one or more of the following conditions:

1. They have no operands in common.

2. One is the logical predecessor of the other.

3. The output operand set of one and the total operand set of the other have a null intersection.

Definition 2: A computation is completely functional if the value history of each of its data words is dependent only on the initial state of the computation, that is, on its input data values.

In particular, a computation will not be completely functional if its data values depend on the relative timing of instructions which can be applicable at the same time. Hence, it would appear that a necessary, and perhaps sufficient, condition for a computation to be completely functional is that the computer system in which it is executed be conflict-free with respect to its instructions. This supposition is indeed true in both respects. For a formal proof of these facts, the reader is referred to the work of Luconi.[5]

The theorem we seek can now be presented as follows:

Theorem 1: The duplicate storage of data words within a first-level memory module will affect neither the presence nor absence of complete functionality for any computation in the computer system.

Proof: Let X be a data word being elevated into first-level memory module M, which already contains a copy of this word, as part of paragraph P. Then four cases can be identified, depending upon whether the old copy of word X has been changed (C) or unchanged (U) since its arrival in M, and whether X is (A) or is not (N) the word accessed by the instruction i causing the paragraphing operation.

Case 1: U, N

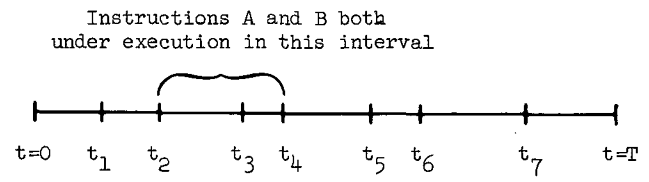
Word X will be immediately stored in module M. Future references to X will retrieve two identical copies, and can destroy either.

Case 2: C, N

Word X will be immediately stored in module M. Future reference to X will retrieve two different copies, only one of which (the old one) will have its change indicator set. Thus the correct value of X is identifiable, and the incorrect one can be destroyed.

Case 3: U, A

Instruction i receives word X, which is correct, directly. If i changes X, future instructions will know that the new copy is correct as only that copy will have its change indicator set. The only problem that could arise would occur if another instruction retrieved the old copy of the data word before the



t_1: instruction A searches for word x, finds it in lower level of memory

t_2: instruction B begins this same search

t_3: instruction A retrieves word x

t_4: instruction A stores the changed value of word x in main memory module m

t_5: instruction B retrieves new copy of word x

t_6: instruction B stores the changed new copy of word x in main memory module m

t_7: instruction C retrieves both copies of word x from module m and notes error - values disagree, both words have change indicator set

Figure 5 - Timing Diagram Showing Conflict that Resulted in Multiple Value Error

revised word X were stored in M. But then this latter instruction and i would violate the conflict-free condition, the computation could not have been completely functional, and the resulting errors would have occurred for any storage system.

Case 4: C, A

Instruction i receives word X, which possesses the incorrect data value. However, the instruction which changed the old copy of X and instruction i must have satisfied the relative timing constraints illustrated in Figure 5, or else paragraphing would not have occurred. Hence, again the computation could not have been completely functional.

QED

In summary, the discussion of this section has indicated that storage reallocation for an associative memory computer system should proceed in the following manner. When a phrase requested by a computation is not found in first-level memory, the paragraph containing this phrase should be read non-destructively from its present location in lower level memory and be stored in the main memory module which is serving as the primary store of the owner of the information contained in the paragraph. It is occasionally possible that this operation will result in two or more copies of a word residing in the same first-level memory module. Should any instruction subsequently retrieve duplicate copies of a memory word, it should proceed as follows:

1. If all copies of the word are identical, it can use any of them and should destroy the others.

2. Otherwise, if only one copy has its change indicator set, this copy should be used and the others destroyed.

3. Finally, if two or more of the copies have their change indicator set, it should have the system print an error message indicating that the relevant computation has a conflict within it.

When a word is subject to removal from main memory, it should be restored in its original position in the hierarchy only if its change indicator has been set, otherwise it may simply be overwritten by a newly elevated word.

Conclusions

Because of the nature of associative memories, paging cannot be used for storage reallocation among their levels. Rather, paragraphs, based on program behavior instead of storage contiguity, are required. Two vastly different types of paragraphs are possible: preset paragraphs, which are formed prior to execution, and dynamic paragraphs, which are formed when accessed. The former type have the advantages of uniqueness, single access retrieval, and simplified replacement rules, while the latter type make a superior choice of words to elevate (at the expense of storage duplication) and allow more selective replacement algorithms to be employed.

The retrieval problems resulting from the use of hierarchical associative memories are due to the need for the multiple storage of words in memory. Assume only one copy of each item existed in the hierarchy. Then if process A, looking for word X, failed to locate that word because process B were simultaneously moving it into main memory, process A would never find word X. The only solution to this problem is to read non-destructively for storage reallocation. But using this multiple storage strategy could result in several copies of a word residing simultaneously in main memory. Fortunately, a simple reallocation scheme will prevent any errors from arising due to this effect.

References

1. Dennis, J. B., "Programming Generality, Parallelism and Computer Architecture," MIT Project MAC, Computation Structures Group Memo No. 32.

2. Gertz, J. L., Hierarchical Associative Memories for Parallel Computation, MIT Project MAC, MAC-TR-69, June 1970.

3. Hanlon, A. G., "Content-Addressable and Associative Memory Systems, A Survey," IEEE Transactions on Electronic Computers, Vol. EC-15, No. 4, August 1966, pp. 509-521.

4. Cannell, M. H., et. al., Concepts and Applications of Computerized Associative Processing, Including an Associative Processing Bibliography, MITRE Corporation, ESD-TR-70-379, December 1970.

5. Luconi, F. L., Asynchronous Computational Structures, MIT Project MAC, MAC-TR-49, February 1968.