A Hybrid Scheme for Processing Data Structures in a Dataflow Environment

Ben Lee, A. R. Hurson, and Behrooz Shirazi, *Member, IEEE*

Abstract— The asynchronous nature of the dataflow model of computation allows the exploitation of maximum inherent parallelism in many application programs. However, before the dataflow model of computation can become a viable alternative to the control flow model of computation, one has to find practical solutions to some major problems such as efficient handling of data structures. This paper introduces a new model for handling data structures in a dataflow environment. The proposed model combines constant time access capabilities of vectors as well as the flexibility inherent in the concept of pointers. This allows a careful balance between copying and sharing to optimize the storage and processing overhead incurred during the operations on data structures. Using simulation, we present a comparative analysis of our model against other data structure models proposed in literature.

Index Terms— Concurrency, dataflow, data structures, parallelism, side-effect free operations, simulation, time and space analysis.

I. Introduction

With the ever increasing demand for high-speed computations, several computer designers have proposed the *dataflow* model as an alternative to the conventional *control flow* model of computation [18], [19], [27], [28], [34], [35]. The attractiveness of the dataflow concept stems from the fact that dataflow operations are asynchronous in nature. That is, an instruction is *fired* (executed) only when all the required operands are available. This is radically different from the control flow method in the sense that the program counter, which is used to sequentially order the instruction execution, is not used.

A dataflow program is represented as a directed graph consisting of *nodes* (or actors) which represent the functions and arcs which represent the data dependencies between the nodes. The operands are conveyed from one node to another in data packets called *tokens*. Therefore, the instructions in dataflow do not impose any constraints on sequencing except the data dependencies in the program.

The dataflow model of computation, due to its distributed control, offers promising improvements in performance [18], [30]. However, before dataflow computers can become a viable alternative to the conventional control flow machines, several major drawbacks must be dealt with. One major problem facing dataflow architecture designers is the handling of *data structures*. This is due to the functionality principle of the dataflow model—i.e., all operations are side-effect free. Therefore, when a scalar operation is performed, new tokens are generated after the input tokens have been consumed. However, if tokens are allowed to carry vectors, arrays, or other structures in general, the absence of side-effects implies that an operation on a structure element must result in an entirely new structure. Although this solution is acceptable from a theoretical point-of-view, it creates excessive overhead at the level of system performance.

The major thrust of this paper is to provide an alternative solution to the problem of handling data structures. The scheme combines both aspects of copying and sharing in order to minimize the storage and processing overhead without introducing a bottleneck due to centralization of accesses. This is achieved by combining constant-time access capabilities of vectors with the flexibility inherent in the concept of pointers (i.e., a hybrid scheme).

Section II provides a brief overview of the methods proposed in the literature for representing arrays. The major implementation characteristics and shortcomings of each method will be outlined. In Section III, the proposed hybrid scheme will be presented. This discussion will also be supplemented by a memory system capable of handling concurrent access to the data structures. Using simulation, a comparative analysis of our hybrid scheme against other proposed methods will be addressed in Section IV.

II. An Overview of Data Structure Representations

There are two basic approaches to representing data structures, such as arrays: *direct* and *indirect* access schemes [15]. In an indirect access scheme, data structures are stored in special memory units and their elements are accessed through explicit "read" and "write" operations. On the other hand, a direct access scheme treats each structure element as individual data tokens—the notion of array is completely removed at the lowest level of computation. The tokens are identified by their tags which carry information about the relative location of the element within an array.

The direct access scheme is compatible with the dataflow principles of execution and it does not require complex structure controller and garbage collection [15], [32]. Naturally, such a concept is directly applicable and efficient for environments where arrays are totally consumed, e.g., matrix multiplication. Moreover, [15] has shown the application of

Manuscript received May 20, 1989; revised April 1, 1991.

B. Lee is with the Department of Electrical and Computer Engineering, Oregon State University, Corvallis, OR 97331.
A. R. Hurson is with the Department of Electrical and Computer Engineering, The Pennsylvania State University, University Park, PA 16802.
B. Shirazi is with the Department of Computer Science, The University of Texas at Arlington Arlington, TX 76019.

IEEE Log Number 9103979.

1045-9219/92$03.00 © 1992 IEEE

such a concept for cases where arrays are partially consumed. In a simulation study of one such proposal, i.e., the token relabeling method, the direct access scheme showed improved performance over the indirect access method in certain application programs [15]. However, the versatility of this concept is an interesting topic which should be studied more extensively.

Despite the above results, literature has shown that storing the data structures and representing them by pointers incurs less overhead than the direct access scheme in transmitting data, reconstructing the resultant array, and randomly accessing array elements. Moreover, in many applications, the notion of array as a single entity cannot be completely done away with—e.g., table look-up applications.

The indirect access approach, however, is not free of obstacles. When array elements are stored in a separate storage, to preserve the functionality principle of dataflow, even a simple operation which modifies an element of an array requires, at least conceptually, copying the entire array except for the element specified by the index. This results in excessive processing and storage overhead. Therefore, one of the major challenges facing dataflow designers is "how should the data structures be represented without compromising the semantics of dataflow, the performance, and the hardware complexities?"

In this section, some of the proposed solutions to the problem of handling data structures will be briefly surveyed. The characteristics of each proposed method will be outlined in order to justify the proposed hybrid scheme.

A. Heaps

In MIT's Static Dataflow Machine, data structures are represented as acyclic directed graphs stored in a separate auxiliary memory [2]. Data structures consist of a set of <selector:value> pairs where a "selector" is an integer or a string and a "value" is any data value including another substructure. The acyclic graphs, or heaps, always form a tree having one root node with the property that each node of the graph can be reached by a directed path from the root node.

The major motivation behind the use of heaps is to alleviate the excessive copying by allowing the common substructures to be shared among several structures. In order to keep track of the number of pointers created and destroyed during the course of a program execution, a *reference count* is associated with each node. When the reference count of a node becomes zero, the node is inaccessible and is placed on a *free storage list* for future usage.

APPEND and SELECT are exclusive operations devoted to manipulating the trees. An append operation creates a new structure which is a version of the input structure containing the modified component. A value can be retrieved from a structure by a SELECT operation. Depending on the selector argument, the corresponding value from the input structure is placed on the output arc. Although tree structures are capable of representing complex data structures, several disadvantages have been identified when arrays are represented as trees.

First, unlike sequentially stored arrays, accessing an element in a tree requires $O(\log n)$ time. Therefore, depending on the depth of the tree, representing arrays as trees may degrade the performance. Second, due to the low level at which the dataflow model of sequencing is applied, operations can only be performed on individual array elements. Therefore, performing two independent append operations on an array still requires sequential execution (i.e., restrained or strict structure). As a result, even simple operations, such as setting a column or a row of a matrix to zero, require the creation of intermediary structures which in turn significantly degrades the performance [12]. Finally, the garbage collection process of unused nodes is a difficult and expensive procedure in the tree representation [2]. This is due to the fact that when the reference count of a node is reduced to zero, the reference counts of the nodes indicated by the pointers in the reclaimed node must also be decreased (i.e., recursion).

B. I-Structures

I-structures were proposed by Arvind and Thomas to provide a data-structuring mechanism for the Tagged-Token Dataflow Architecture (TTDA) [6]. I-structures are asynchronous array-like structures with constraints on their creation and access, i.e., an I-structure is defined once and each of its elements can be written at most once. An I-structure is asynchronous in the sense that the construction of arrays is not strictly ordered; therefore, it is possible for a part of a program to attempt to select an element before that element is appended (i.e., *unrestrained* or *nonstrict* structure).

In order to implement this concept of "consumption before complete production," status bits, called the *presence bits*, are used. The presence bits indicate that an element of a structure has been generated and is ready for consumption. Any requests for an element which has yet to be appended are deferred in a linked list called the *deferred read request list*. When the write request for that location arrives sometime in the future, the presence bits are appropriately updated and the deferred read requests are released.

In spite of improvements over the tree structure method, such as reduced access time and storage requirement, some problems are associated with I-structures. Although I-structures are useful in large class of program constructs where array operations are performed in parallel (e.g., loops), they are not suitable for every programming environment. First, due to the nonstrictness of I-structures, problems arise when modeling certain class of problems which require the mode of production to be strict—e.g., termination of accumulation [8]. In addition, since I-structure elements are appended (written) at most once, there is a potential for race conditions for APPENDs [13].

Since the mid 1980's, the Monsoon Project as an outgrowth of the TTDA has been under investigation. The architecture consists of several highly pipelined processing elements and a set of interleaved memory modules. Processing elements and memory modules are connected via a multistage packet switching network. Processing elements are designed based on the RISC philosophy and memory modules are intended to support I-structure storage as well as imperative storage [11]. Currently a single-node Monsoon prototype is operational and a full scale multiprocessor system is under development [28].

In order to simplify the dynamic dataflow execution and

to remove the need for associative matching unit [30], [32], the architecture of the processing elements is based on the Explicit Token Store (ETS) model—e.g., the notion of active frames. An activation frame provides explicit storage for local operands and special purpose identifiers (e.g., presence bits) to direct instruction processing. Activation frames are similar to the stack frames in supporting imperative languages on conventional machines. However, to accommodate the dynamic nature of the underlying architecture, activation frames are generated as a tree rather than a stack. This concept has also been used in the design of epsilon [18], [19], P-RISC [27], and EM-4 [38]. I-structures and split-phase transactions are employed to facilitate the handling of data structures in Monsoon.

C. The University of Manchester Approach

The data structure representation in the prototype Manchester dataflow computer was proposed by the research group at the University of Manchester [30]. The implementation combines the concept of streams (i.e., a sequence of values communicated between two portions of code [37]) with conventional arrays. However, in contrast to streams, the size of the structure must be known at the time it is created. In this approach a finite component, defined as a collection of finite sequence of elements, is regarded as the "unit" on which the basic storage operations are performed.

There are special operations exclusively used to handle finite component structures. The STORE operation converts a token stream into a stored component and returns a pointer to the component. Note that the STORE operation, which releases the pointer immediately, is unrestrained. A FETCH is the inverse of STORE; it converts a component back into a token stream. The combination of STORE and FETCH operations is used to communicate elements of a data structure between processes. A SELECT operation can be used to randomly access an element of a component. A COLLECT operation is used to determine when all the elements of a component have arrived. This restrained operation ensures that the pointer token is not released until all the elements of a component have been created. The INCREMENT/DECREMENT operation, associated with the reference count scheme, is concerned with garbage collection. Although the University of Manchester design offers advantages brought forth by combining the concept of stream and conventional arrays, selective modification of the element(s) of an array still requires copying the entire array [31].

D. EXtended MANchester Approach

In this approach, an enhanced array structure is used to extend the basic Manchester machine [30]. The machine, referred to as the EXtended MANchester (EXMAN) computer, alleviates the problem of memory overhead by having token pointers pointing to the starting location of an array and increases the access speed by utilizing conventional random access structures.

Initially, an array is sequentially stored as a random access structure. To perform an APPEND operation on an array, a new node is created with its value and index field appropriately filled. The node is then linked to the array and a pointer referencing the new node is returned. This is done by an APPEND operator with three inputs, namely, the array pointer, the index, and the modified value. The SELECT operation is performed by taking the pointer reference given at the input and traversing the linked list until the input index matches the index field of a node. When a match is found, the value field of this node is returned at the output. Otherwise, the required element is obtained from the original array with one more access. To perform a run-time garbage collection on the unused appended nodes, a reference count is associated with each node.

The EXMAN approach avoids excessive copying by utilizing dynamic pointers with conventional random access structures. This in turn leads to a gain in memory space; however, disadvantages still exist. The most obvious problem is due to the number of APPEND operations to an array which consequently affects the search time for SELECT operations.

III. The Proposed Method—Hybrid Structures

In the previous section, several implementation characteristics which directly affect the performance and the memory overhead were discussed. It is apparent from the discussion that an optimum data structure representation requires a careful compromise between several issues. In this section, an alternative solution to data structure representation will be outlined.

In our proposed method, data structures are constructed using a set of *hybrid structures*. A hybrid structure is a union of vector $\boldsymbol{v} = \langle v_0, v_1, \cdots, v_{n-1} \rangle$ and an associated identifier called the *structure template*. Each structure template consists of three fields: left and right pointers and a *descriptor* field. The format for the structure template is shown in Fig. 1. A hybrid structure, therefore, is represented by a structure template with its left *or* right pointer field pointing to a vector $\boldsymbol{v}$ and the descriptor associated with each hybrid structure provides information about the data structure being represented.

The general definition of hybrid structures provided above allows implementation of complex structures by combining constant time access capabilities of vectors as well as the flexibility inherent in the concept of pointers—i.e., a hybrid scheme.

A. Hybrid Structure Representation of Arrays

The basis of our approach in representing arrays[1] is to carefully combine the advantages brought forth by both copying and sharing. The hybrid structure representation also supports strict as well as nonstrict operations on arrays. In our approach, each array is represented by a hybrid structure with its descriptor in the structure template subdivided into three fields: a reference count field, a location field, and a status bit. The *reference count field* (RC) is used to keep track of the number of hybrid structures created and destroyed

[1] Our main discussions of hybrid structures will be based on arrays since their applications are so wide spread in scientific applications. However, this does not preclude other structures being represented as hybrid structures [24].

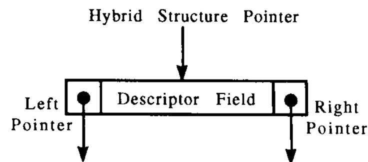


Fig. 1. The basic format of the structure template.

during program execution. The *location field* (LOC) contains a string of 1's and 0's where the length of the string equals the total number of elements in the array. Each location bit determines whether the desired array element resides in the vector indicated by either the left ("0") or right ("1") pointer. Finally, the *status bit* (S) serves a useful purpose in accessing array elements and will be elaborated further in the latter part of this section.

Whenever an array is defined, a hybrid structure is created as follows: the elements of the array are stored sequentially in its vector; the reference count is set to one; the status bit is initialized to zero; the location field is initialized to all 0's; and left pointer field is linked to the vector. Since no modifications have been made, this hybrid structure is considered to represent an original array. Thus, information provided by the descriptor in the hybrid structure indicates that the array has only one reference and all the elements can be accessed from the vector pointed by the left pointer. The basic format of hybrid structure representation of an array is shown in Fig. 2.

Note that since the number of words required for a hybrid structure is known, the pointer fields can be represented as displacements rather than as pointers. This allows constant time access capabilities as in a conventional programming environment. Nevertheless, for illustration purposes, we will refer to them as pointers. In addition, for simplicity, we will limit our discussion of hybrid structures to one-dimensional arrays. This scheme can be extended to multidimensional arrays by representing them in row major or column major ordering.

To allow sharing between the original and modified arrays, they are configured differently. An original array is represented by a hybrid structure with its *left* pointer pointing to a vector (status bit equal to "0"). On the other hand, a modified array is represented by a hybrid structure with its *right* pointer pointing to a vector (status bit equal to "1"). Sharing of array elements between the original and the modified array is achieved by linking the left pointer of the modified hybrid structure back to the hybrid structure containing the original array. The location bits in the descriptor can then be used to access the desired element; ith location bit equal to "0" or "1" indicates that the corresponding array element can be found in the vector linked to the left or right pointer, respectively.

To illustrate our point, consider two operations on arrays—APPEND and SELECT. The APPEND operation takes three inputs (namely an array pointer, an index, and a value) and produces a pointer to a new array where the element indicated by the index is modified. The SELECT operation requires two inputs, an array pointer and an index, and returns a value indicated by the index.

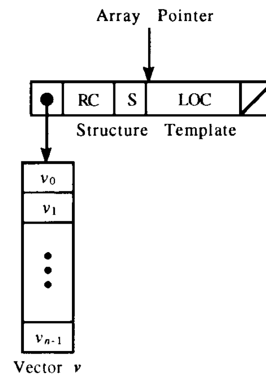


Fig. 2. The proposed array representation.

Consider a simple operation, $B[i] = f(A[i])$, illustrated in Fig. 3. Initiation of this operation first requires the selection of the element a_i from array A—i.e., SELECT. After performing the required function f, an APPEND operation then generates a new hybrid structure with its right pointer pointing to a vector containing only the modified element b_i and its left pointer pointing back to the original hybrid structure.[2] Therefore, for selective updates, the structure template associated with each conceptual array minimizes copying by allowing only the modified elements to be appended to the new array. The pointer can then be used to share the data elements between modified and original arrays.

In cases where an array is entirely updated, the sharing is no longer necessary and the dependence between original and modified arrays can be removed. For example, in Fig. 3, if the array A is totally updated to generate array B, the status of the array B is changed from "modified" to "original." This is done by updating the structure template of array B—the status bit is changed to zero, the reference count of the hybrid structure pointed by the left pointer (i.e., the original array) is decremented by one, the left pointer is modified to point to the vector (containing the elements of array B), and the location bits are all reinitialized to 0's.

Since the basic idea behind the hybrid scheme has been presented, we now describe how the proposed method can be expanded to handle nonstrict operations—such as in I-structures. As discussed in the previous section, I-structures exploit producer–consumer parallelism by allowing the selection of elements before the entire structure is constructed. In order to preserve the determinacy property of the dataflow model, the synchronization of SELECTs and APPENDs is provided by the use of presence bits.

In the proposed method, the same kind of information is embedded in the structure template when an array is defined. Therefore, whenever the compiler detects a construct which is suitable as an I-structure producer or is explicitly declared by the programmer as unrestrained, a hybrid structure is defined with the status bit set to zero, location bits all set to 1's, and

[2] In the cases where the reference counts of arrays to be modified are ones, the APPEND operations are simplified since the elements can be "modified-in-place" without introducing any side-effects.

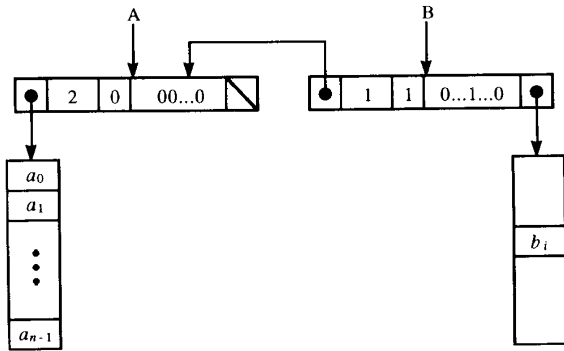


Fig. 3. An example of an APPEND operation.

an empty vector linked to the left pointer. Once an I-structure is allocated, the pointer token is immediately released. As the elements are appended to the I-structure, the corresponding location bit is reset to zero indicating the availability of datum. The synchronization between a producer and a consumer is provided by examining the status bit S and the location bit i to determine how the element i is accessed and whether it is ready for consumption (see Table I). If there is a request to select an element which has not yet been appended, the request is deferred in a linked-list.

TABLE I
INTERPRETATION OF THE S AND i BITS

Bits	Interpretation
00	the data are in the vector pointed by the left pointer
01	the element has not yet been appended
10	
11	the data are in the vector pointed the right pointer

B. Extension to the Proposed Hybrid Method

In the proposed method described thus far, every APPEND operation to an array with a reference count greater than one requires allocation of an entirely new array. This is appropriate if 1) the arrays are small or 2) all or most of the allocated vector elements are eventually used to hold the new values—e.g., updating an entire row or column of an array. Unfortunately, this is not always the case; it is obviously impractical in terms of memory requirements to allocate the entire array for a single random APPEND.

To minimize the memory overhead, the array elements can be partitioned into *blocks*. An *access table* is then used to keep a list of pointers to all the blocks associated with a modified array. Thus, if an n-element array is partitioned into m blocks each containing k elements (e.g., $n = m \times k$), there will be a maximum of m entries in the access table. The displacement within the access table which contains the block pointer and the displacement within the block is determined by using $\lceil i/k \rceil$ and $i \bmod k$, respectively, where i is the array index and k is the number of elements in the block. By utilizing an access table, a search operation for a block which contains the desired element takes a constant time. In addition, the concept of "allocation as required" is employed to reduce the memory overhead. However, one must carefully choose the size of a partition (block) in order to optimize the memory utilization (see Section IV-A).

To illustrate the access table scheme, consider a dataflow graph shown in Fig. 4—three successive APPENDs and a copy operation followed by another APPEND. Fig. 5 shows the hybrid structure after executing the dataflow graph shown

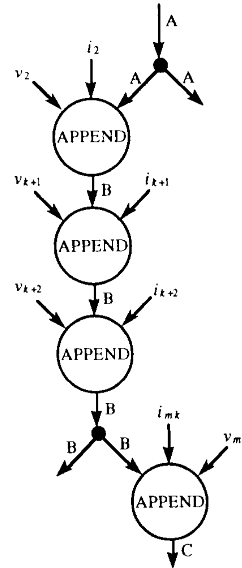


Fig. 4. An example of a dataflow graph.

in Fig. 4. As can be seen, the first APPEND operation, APPEND$[A, i_2, v_2]$, required the allocation of a new hybrid structure (pointed by the array pointer B) with its right pointer pointing to an access table. Since the APPEND operation is performed on the second index of the array, the first entry in the table would be the pointer associated with the first block. The k + 1st element of the array B also has not been allocated; therefore, the second APPEND operation (APPEND$[B, i_{k+1}, v_{k+1}]$) requires adding to the access table another pointer to block of k elements. The third append operation, APPEND$[B, i_{k+2}, v_{k+2}]$, does not require allocation of additional memory space since it belongs to the second block.

The hybrid scheme can also be enhanced to incorporate sharing at the block level. This is achieved by associating an additional reference count, called the *block reference count*, with each block. Hence, when the fourth APPEND operation—APPEND$[S, i_{mk}, v_{mk}]$—is performed, after copying the array pointer B, we can simply share the blocks between the two arrays B and C. This type of sharing at the block level will eliminate any unnecessary copying generally required in performing append operations on a modified structure (status bit equal to "1"). Copying will be limited only to the already appended elements of the block on which the APPEND operation is performed. Moreover, as more allocations are required, the access table will be filled accordingly. SELECT operations

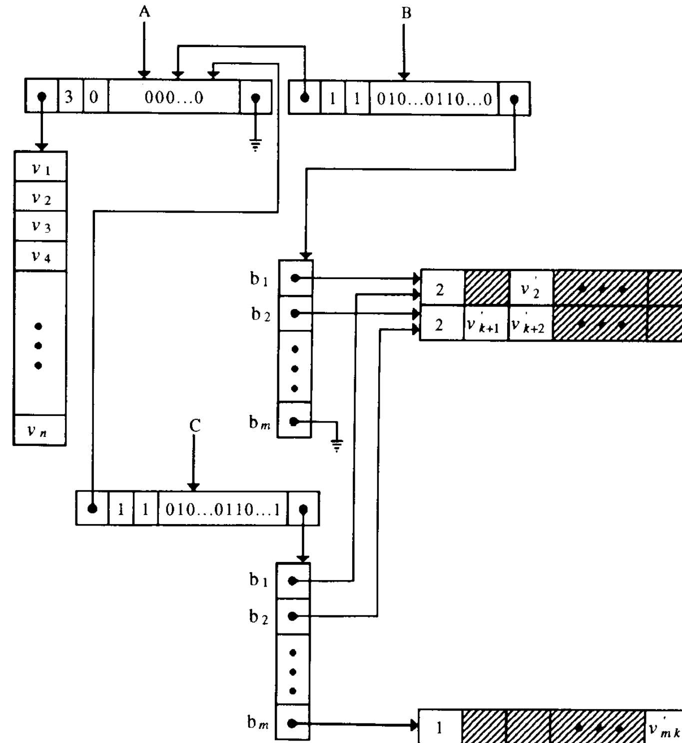


Fig. 5. The access table method with block sharing.

can be performed in a similar manner as described in the previous section. A formal description of the APPEND and SELECT algorithms is given in Algorithms 1 and 2, respectively.

C. The Hybrid Structure Memory System

The basic hybrid scheme for handling data structures has already been discussed. While this solution is conceptually attractive, designing a memory system for efficient handling of concurrent accesses requires careful consideration. The asynchronous nature of array accesses in a dataflow environment requires an appropriate random access parallel memory. The proposed architecture, called the Hybrid Structure Memory Module (HSMM), is shown in Fig. 6. Basically, it consists of a *Structure Template Unit* (STU), an *Access Table Unit* (ATU), and an *Array Storage Unit* (ASU). In order to be truly scalable, we assume that a number of these HSMM's are distributed throughout the dataflow system. For the sake of generality, the discussions of the HSMM's will not pertain to any specific dataflow architecture.

The STU is responsible for handling all incoming messages to the HSMM—APPEND, SELECT, INcrement and DEcrement—which are used to manipulate array elements and to update reference counts. The STU, the ATU, and the ASU, as the names suggest, contain structure templates, access tables, and array elements within each HSMM, respectively. Each unit contains a dedicated processor which manages their own functions. These functions include updating, copying, and processing as well as performing memory allocation and deallocation.

All incoming tokens for the HSMM arrive at the STU. Several different actions can occur at this point. If the token is either <INC, AP> or <DEC, AP>, the STU simply updates the reference count of the array pointed by AP.

A SELECT operation is in the form of <SELECT,AP.i, *dest*>, where i is the index of the array pointed to by AP and *dest* is the destination tag which represents the target node receiving the data. When a select token arrives at the STU, the status bit and the location field of the corresponding structure template is checked to determine where the desired element resides. At the same time, the destination address *dest* is temporarily stored in the STU. If the desired element is contained in the original array, the STU extracts the left pointer *lp* from the structure template and sends a message <read,AP.*lp*.i> directly to the ASU. This is also the case for I-structure reads, which are in the form of <read,AP.i,*dest*>. If the desired data for an I-structure read are not yet available, it is deferred into a link-list.

If the desired element is not in the original array, the STU extracts the right pointer *rp* and sends a message <SELECT,AP.*rp*.i> to the ATU. The ATU determines the access table displacement *atd* (which is equal to $\lceil i/k \rceil$), extracts the selected block pointer *bp*, and sends a message <read,AP.*bp*.i> to the ASU. The ASU then determines the

Algorithm 1: Algorithm for an APPEND operation.

INPUT (array_ptr, index, value);
OUTPUT (new_array_ptr);

/—For nonstrict operations—/
- Store the value in the original_vector$_{\text{index}}$;
- Set new_array_ptr = array_ptr:
- Set the new_array_ptr.loc_field$_{\text{index}}$ to 0;

/—For strict operations—/
IF (array_ptr.ref_count > 1) THEN
- Allocate a new structure template with a new_array_ptr and copy the old structure template pointed by array_ptr;
- Set the new_array_ptr.ref_count to 1;
- Set the new_array_ptr.status_bit to 1;
- Allocate a new access table and link it to new_array_ptr.right_link;
- Allocate a new block and link it to the access table;

  IF (array_ptr.status_bit = 1) THEN
    - Decrement array_ptr.ref_count;
    - Increment new_array_ptr.left_link.ref_count;
    - Copy all nonnull pointers and increment corresponding block ref_count from the old access table (except the new block);
    - If the corresponding old block exists, copy all the existing elements to the new block;

  ELSE
    - Link the new_array_ptr.left_link to the original structure template pointed by the array_ptr;

  ENDIF
  - Store the value in the new block$_{\text{index mod } k}$;

ELSE
  IF (array_ptr.status_bit = 0) THEN
    - Store the value in the original_vector$_{\text{index}}$;
    - Set new_array_ptr = array_ptr;

  ELSE
    IF (block ref_count pointed by access_table$_{\lceil \text{index}/k \rceil}$ > 1) THEN
      - Allocate a new block and copy all the existing values and link it to the access_table$_{\lceil \text{index}/k \rceil}$;
      - Decrement the old block ref_count by 1;

    ENDIF
    - Store the value in the new block$_{\text{index}}$ mod k;

  ENDIF

ENDIF
- Set the new_array_ptr.loc_field$_{\text{index}}$ to 1;

RETURN new_array_ptr;

displacement within the block bd (equal to $i \bmod k$) and reads the desired element. Once the data value v is read, a message <AP,v> is returned to the STU where a token <*dest*,v> is formed and sent to the instruction at *dest*.

Note that all data structure reads are so called split-phase, i.e., the request and reply are not synchronous. Therefore, processors are free to execute any number of enabled dataflow instructions during the roundtrip to HSMM and back.

APPEND request is in the form of <APPEND,AP.i, v,*dest*>. The procedure to append a value to a structure with one reference is similar to that of a SELECT operation, except the final step will be a write operation, e.g., <write,AP.*lp*.i, v> or <write,AP.*bpi*, v>. This is also the case for I-structure writes. If any deferred reads exist, they are processed after the operation.

Appending to a structure with more than one reference requires the creation of a structure template and an access table. Thus, upon reception of a request <APPEND,AP.i, v,*dest*>, the STU processes the corresponding structure template and a series of actions takes place. First, the STU 1) allocates memory for the new structure template, 2) copies contents of the old structure template, 3) updates the left pointer to point to the old structure template, 4) temporarily stores the destination tag *dest*, and 5) sends a message <copy,AP$'$.*rp*.i, v>

Algorithm 2: Algorithm for a select operation.

```
INPUT(array_ptr, index);
OUTPUT(value);

IF (array_ptr.status_bit = 1) THEN
        IF (array_ptr.loc_field_index = 0) THEN
                • Set array_ptr = array_ptr.left_ptr;
                • CALL PROCEDURE CHECK_ORIGINAL(array_ptr)
        ELSE
                • Return (read) the value from the block_(index mod k) pointed by the
                  access_table_⌈index/k⌉;
        ENDIF
ELSE
        • CALL PROCEDURE CHECK_ORIGINAL(array_ptr)
ENDIF

PROCEDURE CHECK_ORIGINAL(array_ptr)
BEGIN
        IF (array_ptr.loc_fieldindex = 0) THEN
                • Return (read) the value from the original_vector_index (and process any deferred requests);
        ELSE
                • The element has not yet been appended. Defer the request in a link-list;
        ENDIF
END
```

to the ATU, where AP′ is the new array pointer.

Second, the ATU 1) allocates memory for the new access table, 2) copies the contents of the old access table pointed to by *rp* (or if the *rp* is nil, as in the case of an original array, no further action takes place), 3) determines the access table displacement *atd*, 4) extracts the selected block pointer *bp*, and 5) sends a message <write,AP′.*bp*.*i*, *v*> to the ASU. Finally, the ASU determines the block displacement *bd* and performs one of the three possible modes of operations: a) allocate a block and append the value (if the *bp* is nil), b) allocate a block, copy existing values from the old block pointed to by *bp*, and append the value (for block reference count greater than one) or, c) simply append the value (for block reference count equal to one).

After the ASU completes the write operation, if the allocation of a new block was required, a message is sent to the ATU in the form of <update,AP′.*bp*′.*i*>, where *bp*′ represents the new block pointer. Then, a message <update,AP′.*rp*′> containing the new right pointer *rp*′ is returned to the STU. As the new pointers are received, each respective unit updates its pointer fields. Once the STU completes its update, the new array pointer is released in the token form of <AP′,*dest*>.

As far as reclaiming storage space is concerned, the decrementing of the block reference count and freeing the blocks occur independently within the submodules. The operation itself is triggered (perhaps after performing a select with a message <DEC,AP>) when the STU detects reference count of zero. The STU returns the structure template to the free storage list as well as sending a message <free,*rp*> to the ATU. In turn the ATU sends a message <free,*bp*> to the ASU.

In order to reduce the amount of INC/DEC operations in the HSMM's, the hybrid scheme exploits two useful features of dataflow languages in managing the reference counts—locality of effect and single assignment rule [3], [4]. The basic idea is to associate a reference count to the structure in a definite "scope" or a block of the program rather than with each individual operation. This way, the reference count can be simply incremented when the hybrid structure is denoted within an active block and decremented when the block is closed.

Note that due to the capabilities of each unit within the HSMM, operations in the different units (STU, ATU, and ASU) are performed in parallel. This significantly enhances the performance of the APPEND operations on the hybrid structure. Moreover, SELECT, INC, and DEC operations can all be pipelined so that several data structure requests can be handled at the same time by the HSMM.

IV. Storage and Time Analysis

Before discussing the storage and time analysis of the proposed hybrid structure, it is important to mention that the results presented here are based on random generation of APPEND and SELECT operations on arrays. The motivation behind such an analysis is to determine the effectiveness of the proposed scheme when selective updates are performed on arrays. In contrast, a loop operation which manipulates a large array does not require the expensive overhead operations of copying the structure template and the access table when the hybrid structure is implemented as I-structures. Nevertheless, the analysis will show that, even under such random

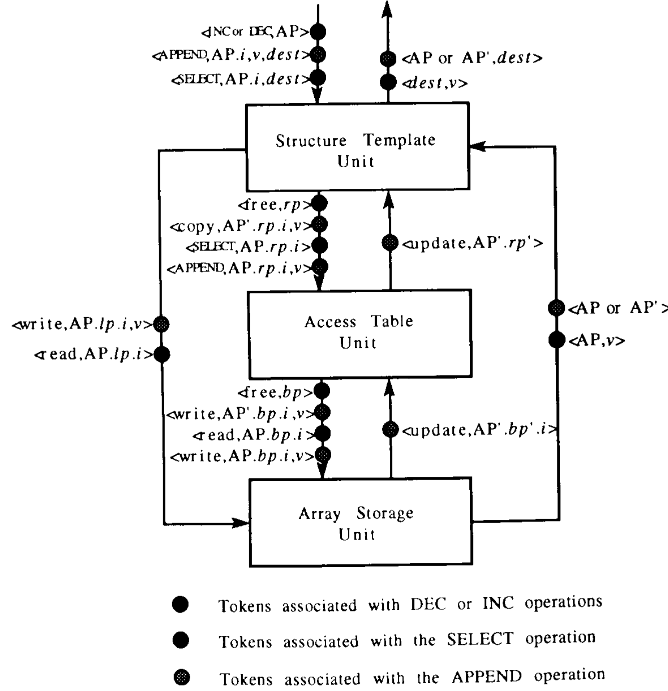


Fig. 6. The hybrid structure memory module (HSMM).

conditions, the hybrid scheme is a promising alternative data structure representation in comparison to other proposed methods in the literature.

The time and storage analysis of the proposed scheme are based on the following assumptions:

1) The hybrid structure is simulated based on the access table scheme using the programming language C.
2) p_1 denotes the probability of an APPEND operation to a structure with a reference count greater than one. This would require the creation of a new structure template, an access table, and a block.
3) p_2 denotes the probability that the APPEND operations require the allocation of an additional block. This situation occurs if an APPEND is performed on a structure with reference count equal to one and the required block has not yet been allocated.

In addition to the assumptions presented above, one implementation change was made to the hybrid scheme. In the original proposed method, the location bit in the structure template indicates whether the desired value is located by following either the left or right pointer. However, associating a bit with each element in a large array will be expensive in terms of storage overhead. Therefore, to reduce the storage requirement, a location bit is associated with an allocated block rather than each element. Of course, this comes at a price of performance, since two additional accesses are needed to check whether the desired value has been appended to the block. The simulation results will show that the advantage gained in space utilization outweighs the cost in performance.

A. Storage Analysis

Since APPEND is an expensive operation, the storage analysis is conducted based on the number of APPENDs performed on the hybrid structure. Moreover, to introduce randomness, we assume that duplication of structures (copy) occurs in between APPEND operations. Word size is assumed to be 32 bit. The following equation is derived for the total size (in words) required to represent hybrid structures based on the number of APPEND operations, σ.

$$S_{\text{HYBRID}} = S_O + \sigma[p_1 S_H + (1 - p_1)p_2 k] \quad \text{where}$$
$$S_O = S_T + n$$
$$S_H = S_T + \lceil n/k \rceil + k.$$

S_T is the basic storage required to store the structure template; i.e., the location field, the reference count field, and the pointer fields. For a wide range of array sizes, the number of words required to store the structure template remains constant (i.e., one word for each field). S_H is the storage requirement to allocate a hybrid structure when an append operation is performed to an array with a reference count greater than one (p_1). The $\lceil n/k \rceil$ term refers to the size of the access table where n and k are the array size and the block size, respectively. S_O is the storage required to represent the original array.

When an APPEND is made to a structure with a reference count of one, three possible situations may occur: 1) the desired block to be appended is already allocated with the block reference count equal to one, 2) the desired block to be

appended is already allocated with the block reference count greater than one, or 3) the append is made on a block which has yet to be allocated. In case 1, no additional storage is necessary. For the case 2, even though the block is already allocated, a block reference count greater than one indicates that a new block must be allocated and all the existing elements must be copied. Finally, in case 3, a new block of k elements is allocated. The allocation of a new block (cases 2 and 3) occurs with a probability of p_2.

One of the major implications on the overall storage requirement of the hybrid structure is the block size k. As k increases, the number of storage elements allocated in response to an APPEND increases while the number of entries in the access table decreases. On the other hand, as k decreases, the block allocation requirement decreases while the size of the access table increases. Therefore, it is important to carefully choose k to optimize the storage utilization. The following equation shows the block size which minimizes the overall storage requirement for the hybrid structure.

$$k = \sqrt{\frac{n}{1 + \frac{(1-p_1)}{p_1} p_2}}.$$

As can be seen, the choice of k is a function of the array size n, p_1, and p_2. Unfortunately, p_1 and p_2 are difficult to predict since they depend primarily on the program environment. For example, p_1 is directly related to the frequency at which the structures are duplicated while p_2 depends not only on the block size k, but also on the number of different versions of the array that currently exist.

The aforementioned equation indicates that an optimal choice for k for various probabilities of p_1 and p_2 lies any where between 1 and $\sqrt{n}$. The simulation results have shown that as p_1 increases, the optimal choice for k approaches $\sqrt{n}$. On the other hand, if p_1 decreases the optimal choice of k also decreases. For the simulated programming environment, p_1 is assumed to be 0.5. The simulation results also indicate that for wide ranges of p_1, n, and k, the average value of p_2 is in the approximate range of 0.85–0.9. With these values, the block size which minimizes the overall storage requirement can be approximated by $k \approx \sqrt{n/2}$. The simulation results of the storage requirement versus block size for various array sizes are shown in Fig. 7. The number of append operations performed was arbitrary chosen to be 60% of the array size and the probability p_1 was kept constant at 0.5. The results verify that optimal choice for k can indeed be approximated as $\sqrt{n/2}$.

For comparison, we can derive the following equations for other proposed methods:

$$S_{\text{COPYING}} = n + \sigma p_1 n$$
$$S_{\text{EXMAN}} = n + 4\sigma$$
$$S_{\text{TREE}} = 3(2^{\lceil \log_2 n \rceil} - 1) + n + \sigma p_1 (3\lceil \log_2 n \rceil + 1).$$

The copying method shows that any append to a structure (reference count greater than one) requires copying the entire structure. This approach represents one end of the spectrum regarding the storage requirement while the EXMAN approach represents the other. In the EXMAN scheme, the storage

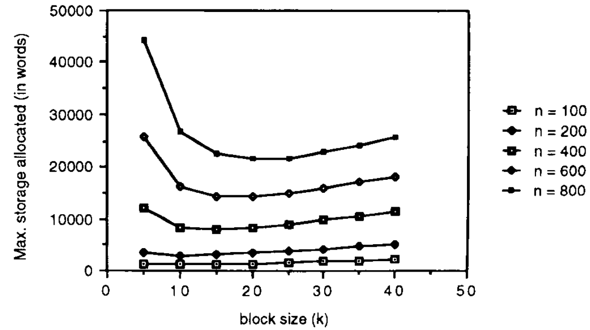


Fig. 7. Storage requirement verus block size k for various array sizes n.

overhead is totally independent of p_1 since a single node containing a reference count, an index, a value, and a pointer is created for any append operation. Finally, the binary tree representation is based on the fact that each append operation requires copying of all the subsequently visited nodes. Each node contains a reference count and two pointers while the leaves contain the actual values.

Fig. 8 shows the relative storage requirement of the hybrid scheme in comparison to the other aforementioned methods. The size of the array is chosen to be 100 and the number of appends performed on the structure is again arbitrarily chosen to be equal to 60% of the array size. As expected, it shows that the copying and the EXMAN scheme offers the worst and best results, respectively. The hybrid scheme shows improvement in comparison to the binary tree method for this particular set of parameters.

In the graph shown in Fig. 9, the probability p_1 was kept constant at 0.5 while the size of the array was varied. In this graph, the copying scheme was deleted to magnify the results of others; however, it is obvious from Fig. 8 that the excessive storage demands of copying cannot compete with the other proposed methods. The results showed that the EXMAN scheme offered the best overall storage requirements. The hybrid scheme, on the other hand, showed improvement over the tree method when the array size was less than 400. As the array size increases, the storage requirement of the hybrid scheme fared less. This is due mainly to the fact that the storage demand for append operations increases logarithmically for trees in contrast to the linear increase for the hybrid scheme. Again, it is important to note that these results are based on random appends only. For constructs, such as loops, the storage overhead associated with the structure template, the access table, and the block is not necessary for every APPEND operation.

B. Time Analysis

The time analysis is based on the following set of assumptions:

1) The number of append and select operations performed on a structure is proportional to 60% and 40% of the array size, respectively.

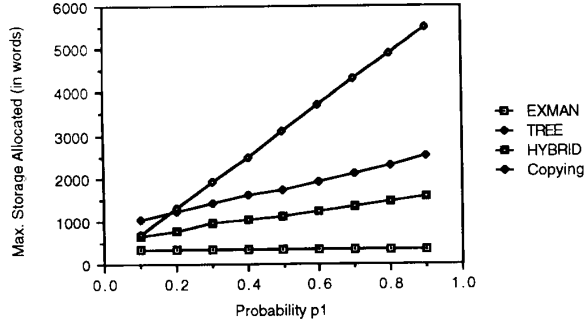


Fig. 8. Storage versus probability p_1 ($n = 100$).

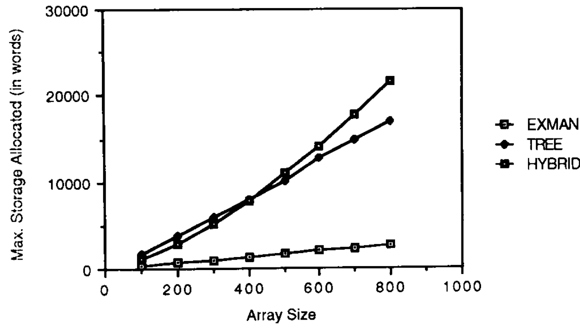


Fig. 9. Storage versus array size n ($p_1 = 0.5$).

2) The select operations are performed upon completion of all the append operations.
3) The performance measure is based on the set of basic operations shown in Table II.

With the assumptions presented above, the following equation can be derived for the average time required to perform an APPEND:

$$T^A_{\text{HYBRID}} = T_{\text{overhead}} + p_1 T_H + (1 - p_1) p_2 T_B + t_w \quad \text{where}$$
$$T_H = 3t_{\text{alloc}} + S_T t_{\text{copy}} + \alpha(t_{\text{copy}} + t_{\text{ref}}) + \beta t_{\text{copy}}$$
$$T_B = t_{\text{alloc}} + p_3 \beta t_{\text{copy}}.$$

The T_{overhead} is the general processing overhead required for append operations. It consists of processing time to initialize the structure template and/or to check on various conditions during the operation. The simulation results indicate that the average overhead time is approximately 11 time units and invariant to the array size or the probability p_1. T_H refers to the time required to generate a hybrid structure when an APPEND is performed to a structure with a reference count greater than one. The variables α and β refer to the *average* number of block pointers and array elements copied, respectively.

If the append operation is performed on an array with a reference count equal to one, a new block must be allocated with the probability p_2. In addition, if the block to be appended has a reference count greater than one, all the existing elements must be copied to the newly allocated block; this occurs with the probability p_3. The simulation results indicate that p_3 increases as the number of appends increases. Finally, one additional access (i.e., t_w) is needed to append the desired value.

TABLE II
TIME DELAYS (IN TIME UNIT) OF THE BASIC SIMULATION OPERATIONS

Symbol	Activity	Time Delay
t_{proc}	perform conditional tests	1
t_{alloc}	allocate a memory block	4
t_{link}	traversing a link	1
t_{copy}	copy a value	2
t_{ref}	update a reference count	1
t_w	a write operation	1
t_r	a read operation	1

As discussed previously, some copying of the block pointers and elements may be necessary whenever append operations are performed. At most, $\lceil n/k \rceil$ pointers and k elements require copying. However, simulation studies have shown that for random APPENDs, the number of pointers (α) and elements (β) copied is quite small. On the average, the actual number of words copied (α plus β) was in the range of 4–8 words for a wide range of array sizes (Fig. 10). More important, the α's and β's are relatively independent of k (Fig. 11). This indicates that minimization of the storage requirement does not affect the overall performance of append operations.

For comparison, the following equations can be derived for the other proposed methods:

$$T^A_{\text{COPYING}} = t_{\text{proc}} + p_1[t_{\text{alloc}} + (n-1)t_{\text{copy}}] + t_w$$
$$T^A_{\text{EXMAN}} = t_{\text{proc}} + t_{\text{alloc}} + 4t_w$$
$$T^A_{\text{TREE}} = (t_{\text{proc}} + t_{\text{link}})\log_2 n + t_w + p_1(t_{\text{alloc}} + 3t_{\text{copy}} + t_{\text{ref}} + 2t_{\text{link}})\log_2 n.$$

The graph illustrated in Fig. 12 shows the average append time versus probability p_1 for $n = 100$. Fig. 13 shows a plot against various array sizes with the probability p_1 kept constant at 0.5. Except for the EXMAN scheme, both graphs show the improvement in performance by employing hybrid structures.

The time analysis for the select operation can be derived in a similar manner. We have assumed that garbage collection is performed during memory system's "idle periods."

$$T^S_{\text{HYBRID}} = T_{\text{overhead}} + t_{\text{ref}} + 2t_{\text{link}} + t_r.$$

Again, T_{overhead} is the average overhead time involved in processing a select operation. The overhead time incurred in the select operation was found to be approximately three time units and is independent of the array size or probability p_1. The select operation requires that the reference count of the structure template be decremented. This is followed by traversing the linked list (mainly the left or right pointer and the block pointer in the access table and/or left pointer of the original structure template) and finally a read operation.

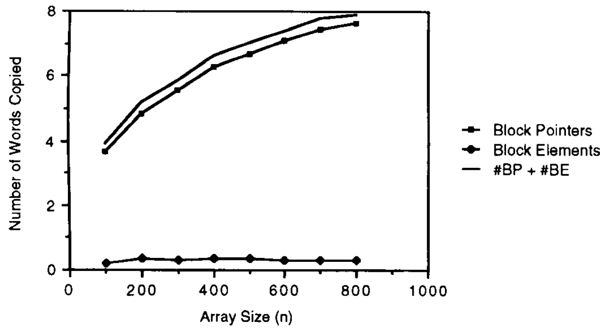


Fig. 10. Average number of words copied versus array size n ($p_1 = 0.5$).

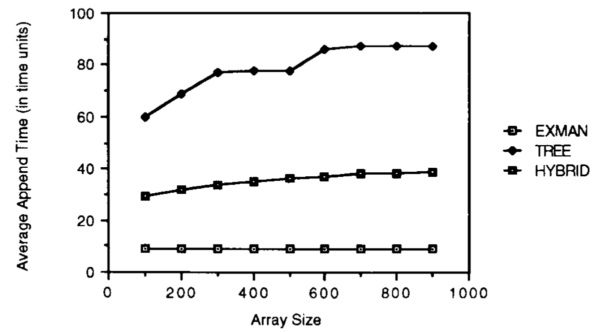


Fig. 13. Average append time versus array size n ($p_1 = 0.5$).

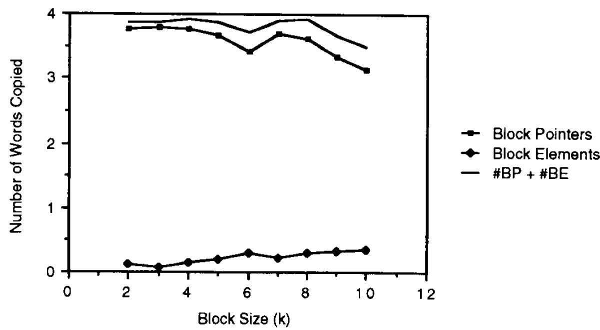


Fig. 11. Average number of words copied versus block size k ($n = 100$).

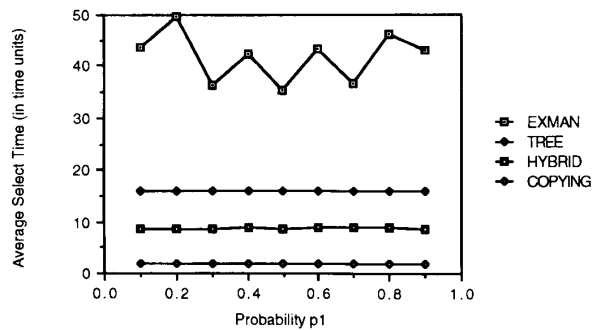


Fig. 14. Average select time versus probability p_1 ($n = 100$).

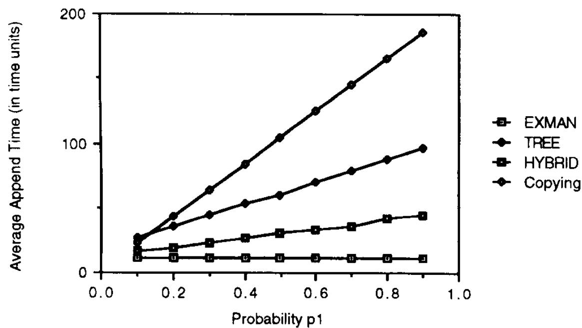


Fig. 12. Average append time versus probability p_1 ($n = 100$).

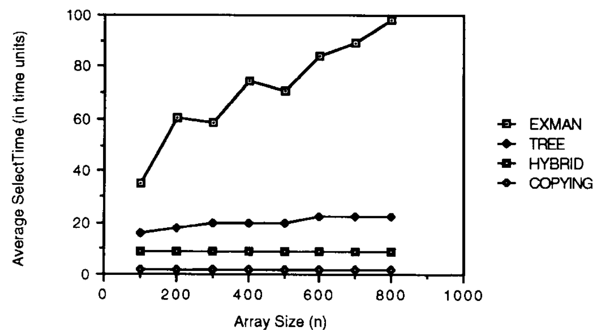


Fig. 15. Average select time versus array size n ($p_1 = 0.5$).

We can derive similar equations to describe performance of select operations for the other methods:

$$T^S_{\text{COPYING}} = t_{\text{ref}} + t_r$$
$$T^S_{\text{EXMAN}} = t_{\text{ref}} + \#\text{link}_{\text{traversed}}(t_r + t_{\text{proc}} + t_{\text{link}})$$
$$T^S_{\text{TREE}} = t_{\text{ref}} + (t_{\text{proc}} + t_{\text{link}})\log_2 n + t_r.$$

The simulation results for the select operation are shown in Figs. 14 and 15. Fig. 14 is a plot of the performance of a select operation versus the probability p_1. The array size was assumed to be 100. As expected, the copying scheme offered the best results. On the other hand, the EXMAN approach exhibited the worst and most erratic behavior. The hybrid scheme showed superior performances over both the tree and the EXMAN approach.

Fig. 15 depicts the execution time versus the array size n. As can be seen, the EXMAN approach also faired poorly in this case. This was mainly due to the increase in the number of appends performed prior to the select operations. The hybrid scheme offered considerable improvement in performance over the tree method.

V. Conclusion and Future Direction

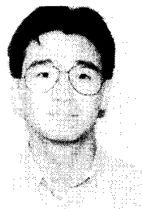
In this paper, an alternative method of representing arrays in a dataflow environment was presented. The hybrid structure is

far from being the ultimate representation. However, it offers flexibility and elegance by carefully combining the advantages brought forth by both copying and sharing. This was achieved by associating a structure template with each vector which provides information about the relationship between an original structure and the modified structures. In addition, the capability of incorporating I-structures enhances the hybrid scheme by offering many of the advantages of unrestrained structures. In order to assert the validity of hybrid structure representation, the storage and time analysis were also presented. Simulation results showed that the hybrid structure offers a promising alternative to the other proposed methods surveyed in this paper. Finally, it also has been shown [23] that the hybrid structure can be used to represent other data structures, e.g., lists and streams.

Our future research efforts will be to integrate the hybrid scheme into dataflow computers. Due to its generality and transportability, hybrid structures and HSMM's can be adopted by existing dataflow systems with minimal design modifications. Furthermore, the comparative analysis of the proposed schemes based on access patterns of existing application programs is an interesting issue which will be studied in the near future. Currently the simulation of an 8-node Monsoon type multiprocessor dataflow machine is under investigation. Among other interesting issues [24], the simulator is intended to investigate the effectiveness of the proposed hybrid scheme against the model proposed in Monsoon Project [11].

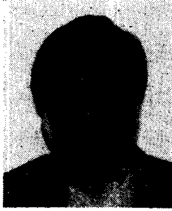
REFERENCES

[1] W. B. Ackerman, "A structure memory for dataflow computers," MIT Lab. for Comput. Sci. Tech. Rep. TR-186, Cambridge, MA, Aug. 1977.
[2] ——, "A structure processing facility for dataflow computers," in *Proc. Int. Conf. Parallel Processing*, Aug. 1978, pp. 166–172.
[3] ——, "Data flow languages," *IEEE Comput. Mag.*, vol. 15, no. 2, pp. 15–23, Feb. 1982.
[4] M. Amamiya, R. Hasegawa, O. Nakamura, and H. Mikami, "A list-processing-oriented dataflow machine architecture," in *Proc. Nat. Comput. Conf.*, 1982, pp. 144–151.
[5] Arvind and D. E. Culler, "Dataflow architectures," *Annu. Rev. in Comput. Sci.*, vol. 1, pp. 225–253, 1986.
[6] Arvind and Thomas, "I-structures: An efficient datatype for functional languages," MIT Lab. for Comput. Sci. Tech. Rep. TM-178, Cambridge, MA, Sept. 1980.
[7] Arvind and V. Kathail, "Multiple processor dataflow machine that supports generalized procedures," in *Proc. 8th Annu. Symp. Comput. Architecture*, May 1981, pp. 291–302.
[8] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: Data structures for parallel computing," in *Proc. Workshop Graph Reduction*, Los Alamos, NM, 1986.
[9] A. P. W. Bohm and J. Sargeant, "Code optimization for tagged-token dataflow machines," *IEEE Trans. Comput.*, vol. 38, pp. 4–14, Jan. 1989.
[10] A. P. W., Bohm, J. R. Gurd, and J. Sargeant, "Hardware and software enhancement of the Manchester dataflow machine," in *Proc. IEEE Compcon*, Spring 1985, pp. 420–423.
[11] D. E. Culler and G. N. Papadopoulos, "The explicit token store," *J. Parallel Distributed Processing*, vol. 10, pp. 289–308, 1990.
[12] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, "A second opinion on data flow machines and languages," *IEEE Comput. Mag.*, pp. 58–69, Feb 1982.
[13] J.-L. Gaudiot, "Structure handling in data-flow systems," *IEEE Trans. Comput.*, vol. C-35, pp. 489–502, June 1986.
[14] J.-L. Gaudiot and M. D. Ercegovac, "A scheme for handling arrays in data-flow system," in *Proc. 3rd Int. Conf. Distributed Comput. Syst.*, Fort Lauderdale, FL, Oct. 1982.
[15] J.-L. Gaudiot and Y. H., Wei, "Token relabeling in a tagged token dataflow architecture," *IEEE Trans. Comput.*, vol. 38, pp. 1225–1239, Sept. 1989.
[16] K. P. Gostelow and R. E. Thomas, "A view of dataflow," in *AFIPS Conf. Proc.*, vol. 48, June 1979, pp. 629–636.
[17] ——, "Performance of a simulated dataflow computer," *IEEE Trans. Comput.*, vol. C-29, pp. 905–919, Oct. 1980.
[18] V. G. Grafe *et al.*, "The epsilon dataflow processor," in *Proc. 16th Annu. Int. Symp. Comput. Architecture*, 1989, pp. 36–45.
[19] V. G. Grafe and J. E. Hoch, "Implementation of the epsilon dataflow processor," in *Proc. 23rd Hawaii Int. Conf.*, 1990, pp. 19–29.
[20] A. R. Hurson, B. Lee, and B. Shirazi, "Hybrid structure: A scheme for handling data structures in a data flow environment," in *Proc. Parallel Architectures and Languages, Europe '89*, June 1989, pp. 323–340.
[21] K. Kawakami and J. R. Gurd, "A scalable dataflow structure store," in *Proc. Symp. Comput. Architecture '86*, pp. 243–250.
[22] R. M. Keller, "Divide and CONCer: Data structuring in applicative multiprocessing systems," in *Conf. Rec. 1980 ACM Lisp Conf.*, 1980, pp. 196–202.
[23] B. Lee and A. R. Hurson, "Data structure handling in data flow," Tech. Rep. TR-88-059, Dep. Elec. Eng., The Pennsylvania State Univ.
[24] B. Lee, A. R. Hurson, and T.-Y. Feng, "A vertically layered allocation scheme for data flow systems," *J. Parallel Distributed Processing*, vol. 11, pp. 175–187, 1991.
[25] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960.
[26] D. Misunas, "Structure processing in data-flow computer," in *Proc. Sagamore Comput. Conf. Parallel Processing*, 1975, pp. 230–234.
[27] R. S. Nikhil and Arvind, "Can dataflow subsume von Neumann computing?," in *Proc. 16th Annu. Int. Symp. Comput. Architecture*, 1989, pp. 262–272.
[28] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token store architecture," in *Proc. 17th Annu. Int. Symp. Comput. Architecture*, May 1990.
[29] G. M. Papadopoulos, "Program development and performance monitoring on the Monsoon dataflow multiprocessor," Tech. Rep. TR-303, MIT Lab. for Comput. Sci., 545 Tech. Sq., Cambridge, MA 02139, Oct. 1989.
[30] L. M. Patnaik, R. Govindarajan, and N. S. Ramadoss, "Design and performance evaluation of EXMAN: An EXtended MANchester data flow computer," *IEEE Trans. Comput.*, vol. C-35, pp. 229–243, Mar. 1986.
[31] S. Samet and M. Gokhale, "Data structures on data flow computers: Implementations and problems," *IEEE 1984 Micro-Delcon*, pp. 84–96.
[32] J. Sargeant and C. C. Kirkham, "Stored data structures on the Manchester data flow machine," in *Proc. Symp. Comput. Architecture '86*, pp. 235–242.
[33] A. J. Smith, "Multiprocessor memory organization and memory interference," *Commun. ACM*, vol. 20, no. 20, pp. 754–761, Oct. 1977.
[34] V. P. Srini, "An architectural comparison of data flow systems," *IEEE Comput. Mag*, vol. 19, pp. 68–86, Mar. 1986.
[35] A. H. Veen, "Dataflow machine architecture," *ACM Comput. Surveys*, vol. 18, no. 4, Dec. 1986.
[36] I. Watson and J. R. Gurd, "A practical data flow computer," *IEEE Comput. Mag*, vol. 15, pp. 51–57, Feb. 1982.
[37] K. S. Weng, "Stream-oriented computation in recursive data flow schemas," MIT Lab. for Comput. Sci. Tech. Rep. TR-68, Cambridge, MA, Oct. 1975.
[38] Y. Yamaguchi, S. Sakai, K. Hiraki, Y. Kodama, and T. Yuba, "An architectural design of a highly parallel dataflow machine," *Inform. Processing*, pp. 1155–1160, 1989.



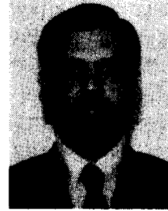
Ben Lee received the B.E. degree in electrical engineering from the State University of New York (SUNY) at Stony Brook, New York, in 1984 and received the Ph.D. degree in computer engineering from The Pennsylvania State University, University Park, in 1991.

He is currently an Assistant Professor of Computer Engineering at Oregon State University. Since 1985, as a graduate student, he has taught various electrical and computer engineering courses, including a course in computer system architecture. He has also published a number of papers in the area of dataflow architecture, task scheduling, and VLSI. His interests are in computer architecture, parallel processing, dataflow architecture, and VLSI.



A. R. Hurson is a Computer Engineering Faculty member at the Pennsylvania State University. His research for the last 10 years has been directed toward the design and analysis of general purpose as well as special purpose computer architectures. He has published over 100 technical papers in areas including computer architecture, parallel processing, database machines, dataflow architectures, and VLSI algorithms. He served as Co-Guest Editor of a Special Issue of the IEEE PROCEEDINGS on Supercomputer Technology and is the co-author of an IEEE Tutorial on Parallel Architectures for Database Systems.

Dr. Hurson has been active in various conferences, and has given tutorials for various conferences on database management systems, supercomputer technology, data/knowledge-based systems, and parallel computing. He is a member of the IEEE Computer Society Press Editorial Board and a member of the IEEE Distinguished Visitor Program.



Behrooz Shirazi (S'83–M'85) received the Ph.D. degree in computer science from the University of Oklahoma, in 1985.

He is currently an Associate Professor of Computer Science Engineering at The University of Texas at Arlington. Prior to that he was an Assistant Professor of Computer Science and Engineering at Southern Methodist University from 1985 to 1990. His research interests and expertise encompass parallel and distributed computing systems, task allocation and load balancing, heterogeneous distributed computations, and dataflow systems. He has over 50 technical publications in the above areas.

Dr. Shirazi founded the IEEE Symposium on Parallel and Distributed Processing in 1989. He has been the Chair of the Dallas Chapter of the IEEE Computer Society as well as the IEEE Region 5 Area Activities Board Chair. He is a member of the IEEE Computer Society and the Association for Computing Machinery.