A list-processing-oriented data flow machine architecture

by MAKOTO AMAMIYA, RYUZO HASEGAWA, OSAMU NAKAMURA, and HIROHIDE MIKAMI
Musashino Electrical Communication Laboratory, N.T.T.
Tokyo, Japan

ABSTRACT

This paper analyzes some issues concerning list processing under a data flow control environment from the viewpoint of parallelism and also presents a new type of list-processing-oriented data flow machine, based on an association memory and logic-in-memory.

The mechanism of partial execution in each function is shown by example to be effective in exploiting the parallelism in list processing. The lenient cons mechanism is shown to exploit maximally parallelism among activated functions.

1. INTRODUCTION

A data flow machine, whose basic idea was offered by J. B. Dennis[1] and for which several research efforts are being pursued at several places in the world,[2–6] is a very attractive concept as a future computer architecture, from the following viewpoints:

1. A data flow machine exploits the parallelism inherent in problems.
2. Recent noteworthy advances in VLSI technology have been made. A data flow machine makes effective use of numerous VLSI devices and makes possible the implementation of a distributed control mechanism.
3. Functional programming will become increasingly important to the improvement of software productivity. A data flow machine effectively executes programs written in a functional language.
4. Nondeterministic execution[7] will become an important mechanism in future computer systems. A data flow machine is expected to execute nondeterministic programs effectively because of its parallelism.

However, many problems remain to be solved in order to achieve an actual data flow machine in a real environment. Especially when considering Items 3 and 4 just cited, it is necessary to clarify the data flow machine's applicability to nonnumerical problems.

This paper discusses list processing, which is typical of nonnumerical data processing, on a data flow machine, keeping the Lisp data structure and operations in mind. The main reasons why Lisp was considered are that Lisp has a simple and transparent data structure and that it contains the basic problems in structured data manipulation.

First, parallelism in list processing is discussed, and it is pointed out that this can be achieved by parallel evaluation of function arguments and partial execution of the function body. Then it is shown that parallelism increases dramatically with introduction of a lenient cons concept into the data flow execution control. Next, list-processing-oriented data flow machine architecture and structure memory construction methods are presented. Finally, a garbage collection algorithm, based on the reference count method, is discussed.

All programs throughout this paper are described in VALID[8] language, which is designed as a high-level programming language for the data flow machine presented in this paper.

2. LIST PROCESSING UNDER A DATA FLOW CONTROL ENVIRONMENT

The noteworthy data flow execution control effects are as follows:

1. It exploits the maximal parallelism inherent in a given program both on a low level (primitive operation level) and on a high level (function activation level).
2. It effectively executes programs constructed on the basis of the concept of functional programming, which has no notion of program variables and side effects (i.e., rewriting the global variables).

The parallelism of the primitive operation level is achieved by the data-driven control principle; that is, each operation is initiated without attention to other operations when all of its operands have arrived. Function-activation-level parallelism is obtained by the partial evaluation mechanism:

1. Each argument of a function is evaluated concurrently.
2. The execution of a function is initiated when one of the arguments of the function is evaluated, and the caller function resumes its execution when one of the return values is obtained in the invoked function execution.

In this section these parallel execution mechanisms are examined through several examples.

2.1. Parallel Evaluation of Arguments

Programs written in VALID are transformed to equivalent pure functional representation, i.e., the form of prefix notation, and equally translated to data flow graphs. For instance, Program1, which reverses a given list in each level, is translated by the VALID compiler into the data flow graph shown in Figure 1. Block1 in Program1 is equivalently represented in the prefix notation

fulrev(cdr(x), cons(fulrev(car(x), nil), y)).

In this expression the two arguments cdr(x) and cons(...) for the function fulrev are evaluated in parallel; and before evaluating the argument cons(...), its two arguments fulrev(...) and y are evaluated in parallel, and so on. Thus, the evaluation of a function, in general, proceeds from the inner to the outer (i.e., innermost evaluation). This results in highly parallel evaluation of the innermost arguments. In other words, each evaluation is independent of the other evaluations under the condition that the evaluation is initiated only when all values of arguments are obtained (which is called data-driven control).

Program1—Mirror image of tree
fulrev: function (x,y) return (list)

```
= case
  null(x) →y;
  atom(x)→x;
  others →clause
          ⌈ u = cdr(x);
          | v = fulrev(car(x),nil);
  block1  | w = cons(v,y);
          ⌊ return fulrev(u,w)
            end
end
```

2.2. *Partial Execution of Function Body*

The parallelism, based on the parallel evaluation of arguments for each function, is limited because the nesting of arguments is limited in source text. This restriction on parallelism, however, can be overcome by executing the function body partially.

If the data-driven control principle is applied to the function activation, as in the case of primitive operations, every function is activated only after all its arguments are evaluated. In this case, time is wasted unnecessarily in each function activation through waiting for the completion of all its argument evaluations. However, if each value is passed into the function body immediately when it is evaluated, and the function body execution proceeds partially every time the value is passed in, efficient execution can be obtained, because the unnecessary waiting is cut out at function activation time.

The function activation and argument-passing mechanism for the partial function execution is implemented as shown in Figure 2. The data flow graph in Figure 2(c) represents the activation control for the function

$$[y1, y2, \ldots, yn] = f(x1, x2, \ldots xm).$$

The call node, which creates a new environment for the activated function, is initiated by the "or" gating nodes, when one of the tokens (values) has arrived. Here, the call node creates the body first if the body does not exist. Otherwise, it creates only an instantiation name. The "or" gate implementation uses a t/f switch, as shown in Figure 2(b).

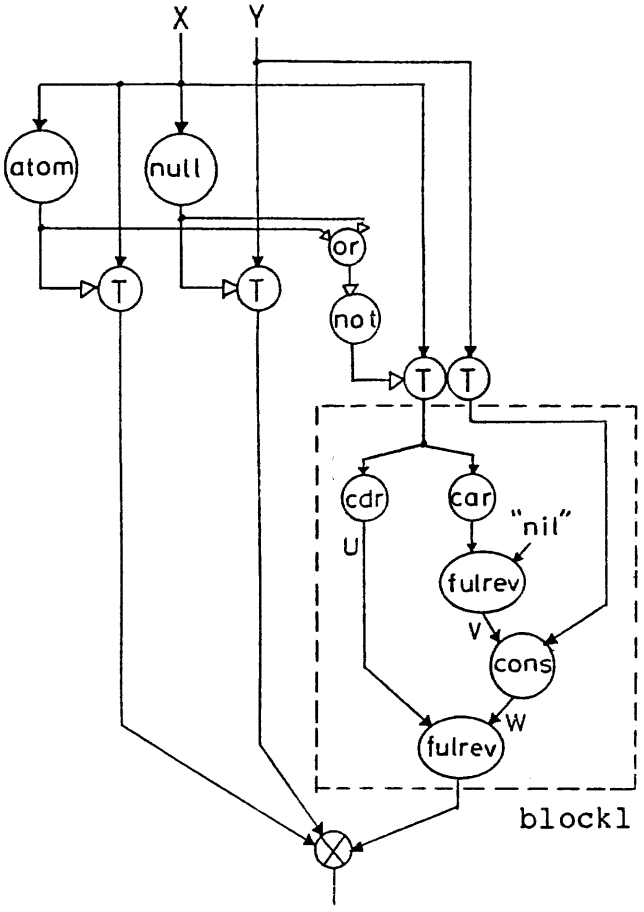
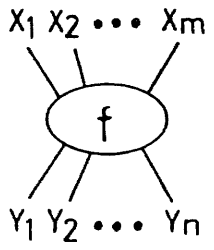
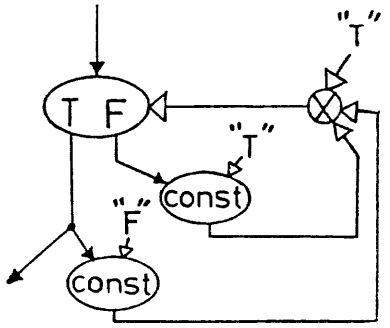


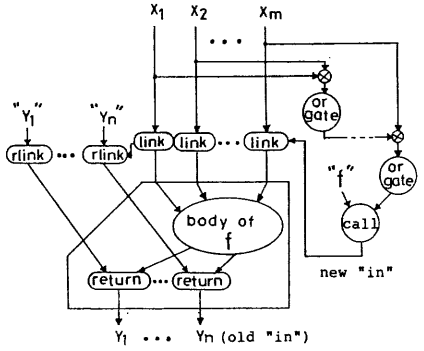
Figure 1—Data flow graph for fulrev



(a) Function invocation node



(b) Or gate implementation



(c) Function activation control

Figure 2—Function activation mechanism

When the new environment is created and the body is ready to run, the token "in" (instantiation name of the activated function) is sent to link nodes and rlink nodes. Each link node passes each argument value x1, x2, . . . , xm to the body of the activated function every time each value has arrived. Each rlink node passes information regarding the place where the return value is sent. These bits of information y1′, y2′, . . . , yn′, each of which is determined at compilation time corresponding to y1, y2, . . . , yn, are attached to each return value to identify its destination. As each return value is passed back to the calling function as soon as it is generated, the calling function can resume and proceed with the execution partially every time the return value is passed back from the called function. Here each function is permitted to return multiple values (i.e., the tuple of values) under the data flow environment.

2.3. Lenient Cons and Parallelism by Pipelined Processing

Although the partial execution of function yields higher parallelism, it is not sufficient for maximally exploiting the parallelism inherent in the given program.

In program2, for instance, the function partition in sort body divides a list into three lists, y1,y2,y3, each of which contains elements less than, equal to, and greater than the first element. As the sort and append are activated immediately after each of y1,y2,y3 is generated, it is expected that the maximal parallelism among functions is obtained. However, parallelism by partial execution of the function body does not work well for reducing the execution time in the order, since the time spent to sort the list of length n is proportional to the square of n in the worst case. (Though it is proportional to n in the best case.) The reason is that since each of the values y1, y2, and y3 is not returned until the append operation is completed in the partition body, the execution of the sort function, which uses those values, must wait until they are returned, and the waiting time is proportional to the length of the list data made by the append operation.

```
program2—Quicksort program
sort: function (x) return (list)
= if x=nil then x
          else clause
               y = list(car(x));
                 [y1,y2,y3] = partition(cdr(x),y);
                 return
                 append(sort(y1),append(y2,sort(y3)))
               end;
partition: function (x,y) return(list,list,list)
= if x=nil then (nil,y,nil)
          else clause
               [w1,w2,w3] = partition(cdr(x),y);
               x1 = car(x); y1 = car(y);
               return
                 case
                   x1=y1→(w1,append(list(x1),w2),w3);
                   x1<y1→(append(list(x1),w1),w2,w3);
                   x1>y1→(w1,w2,append(list(x1),w3))
                 end
               end;
```

If the former parts of the list, which are partially generated, are returned in advance during the period when the latter parts are appended, the execution which uses the former parts of the list can proceed. Thus the producer and the consumer executions overlap each other. As the append is the repeated application of cons, as Program3 shows, this problem can be solved by introducing leniency into the cons operation.

```
Program3—append
append: function (x y) return (list)
= if x=nil then y
          else cons(car(x),append(cdr(x),y))
```

Lenient cons, which is slightly different from the idea of "suspended cons,"[9] means the following: For the operation of cons(x,y), the cons operator creates a new cell and returns its address as a value in advance before its operand x or y arrives. Then the x and y values are written in the car and cdr field of the cell, respectively, when each of them has arrived at the cons node.

In the implementation the cons operator is decomposed into three primitive operators, getcell, writecar and writecdr, as shown in Figure 3. The getcell node is initiated on the arrival of a signal token, which is delivered when the new environment surrounding the cons operation is created. The getcell operator creates a new cell and sends its address to the writecar node, the writecdr node, and the nodes waiting for that cons value.

Each memory cell has, in addition to the garbage tag, the car-ready tag and the cdr-ready tag, each of which controls read accesses to the car field and the cdr field. The getcell operator resets both ready tags to inhibit read accesses. The

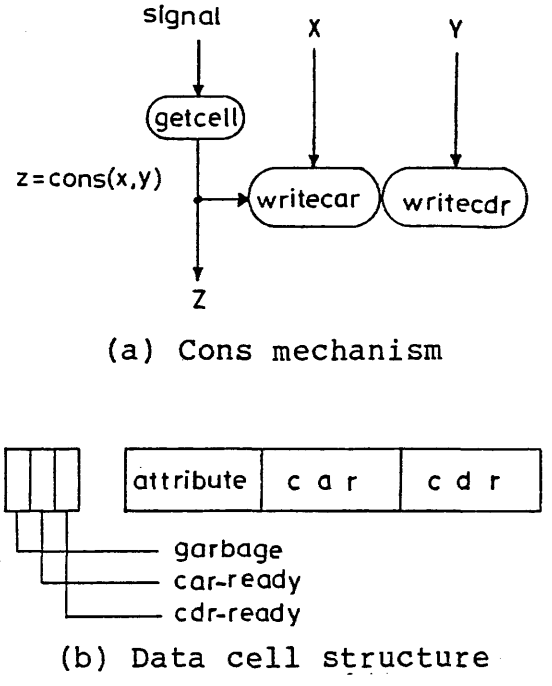


Figure 3—Lenient cons implementation

writecar (or writecdr) operator writes the value x (or y) to the car field (or cdr field), and sets the ready tag to allow read accesses to the field.

Lenient cons has a great effect in list processing. It naturally implements the stream processing feature, in which each list item is processed as a stream[4,10] for programs that are normally written according to the list processing concept, without the notion of stream.

3. DATA FLOW MACHINE ARCHITECTURE

The data flow machine is composed of five components: control modules (CMs), an inter-CM communication network (CN), structure memories (SMs), an arbitration network (AN), and a distribution network (DN), as shown in Figure 4. The CM, which is the kernel of data flow execution control, consists of a memory for data flow machine instructions and the enabled instruction fetch mechanism. The CN connects CMs with each other. The SMs store structured data such as list data. The AN and DN connect CMs and SMs.

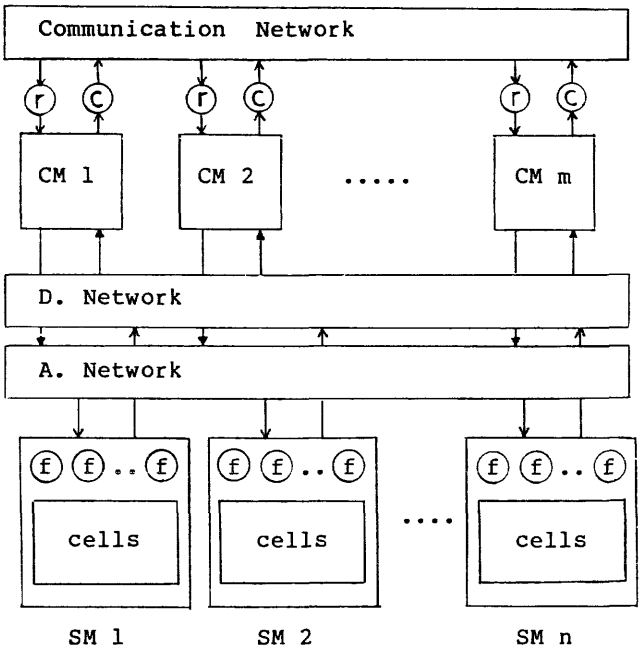


Figure 4—Data flow machine organization

The characteristics of this machine architecture, which is mainly based on the associative memory concept, are as follows:

1. Effective memory utilization can be achieved as a result of dividing the CM memory into instruction memory (IM) and operand memory (OM). The IM, which is a read-only associative memory, contains data flow program (i.e., function body). Here, destination instructions that await a result value are retrieved associatively. The OM acts as a buffer for arriving operands.
2. As function bodies of a program are distributed in each of the CMs, and each CM controls the execution of each function body concurrently, parallelism is achieved among CMs. The call/return parameters among functions are passed through the CN, which logically realizes dynamic tree structure.
3. Operation units are embedded in structure memory. The structure memory is composed of a number of banks, in each of which structured data operation units are equipped.
4. The AN and DN provide paths between CMs and SMs. The AN decodes the operand address in the instruction packet and sends the packet to the addressed SM bank. The DN accepts the result packet, which contains the destination CM address, from SM, and delivers it to the specified CM. The AN and DN are constructed using routing network technique.

This data flow machine architecture can exploit high parallelism due to the concurrent executions among IMs and the pipelined processing between IM and SM.

4. EXECUTION CONTROL

The CM memory which contains data flow machine codes is composed of an IM and an OM, as mentioned before. The IM and OM organization is shown in Figure 5.

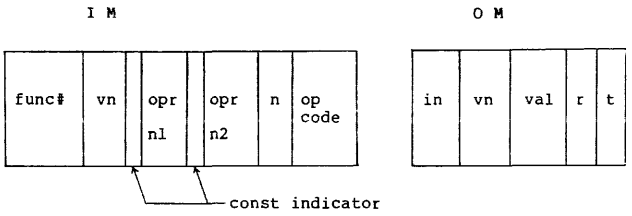


Figure 5—IM and OM field organization

Each memory cell of the IM consists of several fields, function name field (func#), value name field (vn), first operand name field (oprn1), second operand name field (oprn2), operand number field (n), and operation code field (opcode). The OM consists of five fields, instantiation name field (in), value name field (vn), operand value field (val), first/second operand indicator (r), and garbage tag (t). The instantiation name is assigned to a result value so as to share the function body.

The mechanism to deliver result value and fetch an enabled instruction is shown in Figure 6.

When a result packet has arrived at the IM, the func# and oprn1 or oprn2 are examined associatively, using the key ⟨func#, vn⟩, both of which are extracted from the result packet as a search key. If the matched instruction is a one-operand type, an instruction packet is immediately constructed from the matched instruction code and the result value contained in the result packet and sent to the AN.

If the matched instruction is a two-operand type, on the other hand, the in and vn field in the OM are examined for

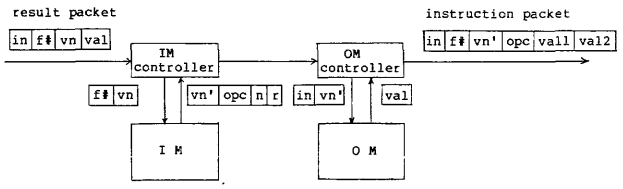


Figure 6—Executive control mechanism

matching associatively against the key ⟨in⟩ in the result packet and the key ⟨vn⟩ extracted from the matched instruction at IM.

If an OM cell is matched, which means one of the two operands has arrived already, the matched data are read out from OM. Then a two-operand type of instruction packet is constructed along with the operand value contained in the result packet and sent to the AN.

If no OM cells are matched, the garbage tag field is accessed associatively to find a free cell. Then, the ⟨in, vn, val⟩ in the result packet and tag r, which indicates whether the val is the first operand or the second operand, are written into the cell taken out.

5. STRUCTURE MEMORY

The method of structured data manipulation is an important problem in the data flow machine architecture.[4] In this section, structure memory design philosophy and its construction method are described from the viewpoint of parallel list processing. In this data flow machine, list structured data are stored in the structure memory, and their pointers to each entry flow in the machine as data tokens.

5.1. Primitive Operation in List Processing and Memory Function

Pure Lisp primitive operations that have no side effect are considered as a basis for structured data manipulation. Among the five primitive operations (cons, car, cdr, atom, and eq), only the cons operation creates a new data cell and writes car and cdr pointer into the cell. Once the value is written into the cell, its contents are never modified. As other operations only refer to the cell, and as programs composed of these five functions have no side effects, the new cell may be created at any location.

List processing is regarded as memory operations which mainly contain readout operations. How to execute the memory operation effectively is a key problem. Memory contention and side effects are serious for exploiting the parallelism in list processing. The parallel execution among memory operations is obtained by preserving functionality, as in pure Lisp.

The data-driven control makes possible the pipelined processing between execution control and memory operation. If the pipe capacity is large enough, execution control is not affected by memory access overhead. Therefore, uninterrupted access to memory cells is possible.

As a new cell may be created by cons at any location, the problem of memory contention can be solved by dividing the structure memory into many banks. In addition, parallelism among memory operations is obtained by providing an operation unit for each memory cell. This idea results in a logic-in-memory concept. When the tradeoff between parallelism and cost is considered, it can be decided whether to embed the operation in a memory device.

5.2. Garbage Collection

As many data elements are copied in the course of the side-effect free data manipulation, how to use structure memory cells effectively is an important problem. Although mark-scan methods are generally used as a garbage collection method in a conventional machine, a reference count method is adopted here, for the following reasons:

1. Since pointers to list data entries are scattered in various parts of the machine, such as instruction memory units, operation units and networks,[10,11] it is very difficult to extract the active cell without suspending execution.
2. As list manipulations have no side effect, no circular lists exist.

In the reference count method, each structure memory cell or memory block has a reference counter field which is updated every time operations, such as car, cdr, etc., are performed. Reference count handling overhead will be serious if the reference count is updated in not only primitive operations but also in T/F switch and function linkage operations. However, this problem can be solved by reducing the reference count update frequency. The method adopted here makes use of VALID language features, that is, (1) block structure and locality of value name, (2) uniqueness of the value name definition (single assignment rule). The reference count management explicitly updates the reference number of the cell by performing the increment and decrement operations. It is not necessary to update the reference number of cells referenced in a block every time operations are performed. Instead, the reference number of the cell which is newly denoted in a block is incremented when the block is opened and decremented when the block is closed.

5.3. Structure Memory Organization

Unlike numerical processing, which handles regular data structures such as vectors and arrays, it cannot be expected that manipulating list-structured data yields locality of access to each list item, since many functions refer to sublists or superlists of a list which is produced by some function, and the sublists and superlists are produced variously during the execution of many functions. In such a case, whether to achieve the locality of access in each function or to distribute access without copying sublist is a tradeoff point in design.

The copying overhead is serious in list processing, because many sublists and superlists are produced in various places in an execution. Therefore, distributing access to lists thoroughly is more effective than copying lists in the data flow machine architecture. New cells are generated in such a way

as to distribute cells uniformly in SM banks, since appropriate cons strategy enables each cell address to be distributed, due to the functionality of list processing, as mentioned above.

The structure memory is composed of a number of memory banks which can control access independently, as shown in Figure 7. The SM bank construction can resolve the memory access bottleneck, because new cells are taken out and distributed uniformly in each SM bank. The reference count management module (cleanup) for garbage collection is provided in each SM bank. As the reference count method is adopted as described above, the function such as a logic-in-memory is required in order to solve the neck of the reference count update operation.

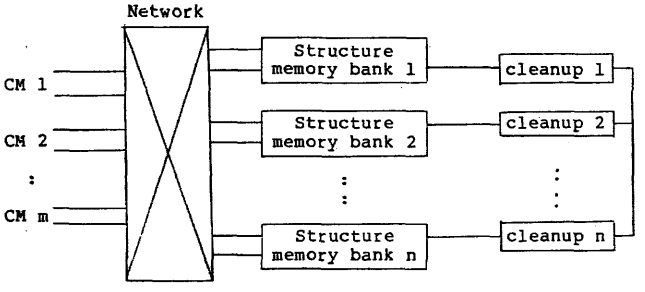


Figure 7—SM structure

The SM bank organization is shown in Figure 8. Data cells in an SM are constructed of three independent blocks, ref, car, and cdr blocks, so as to enhance the primitive-operation-level parallelism. The car(cdr) block consists of car(cdr) ready tag, attr1(attr2) field and car(cdr) pointer field. The attr1 (attr2) field indicates the attribute of the cell pointed by car (cdr) field, i.e., number atom or literal atom or nonatom. Attribute information extracted from the field is also held in an instruction and result packet. The ref block consists of garbage tag and reference counter field which holds the reference number. The ref block is implemented with RAM incorporating the increment and decrement circuits. (The increment and decrement functions are integrated in the memory, based on logic-in-memory concept, so as to reduce the reference count handling overhead in garbage collection management.)

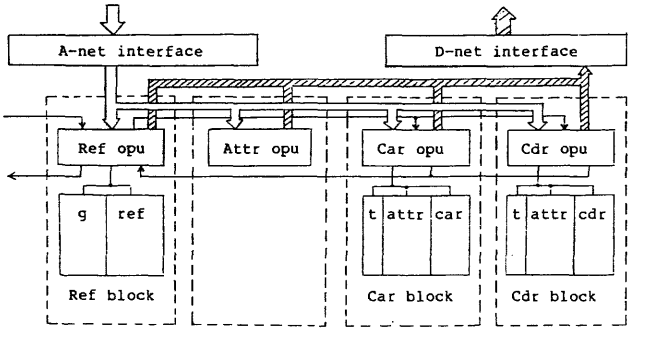


Figure 8—SM bank organization

Specialized operation units are devised for each primitive operation according to the field (i.e., car, cdr, attr and ref) accessed by their operations, as shown in Table I. Car(Cdr) opn performs operations which read the car(cdr) field. Attr opn performs operations which examine the attribute data. Ref opn controls reference count management and performs the getcell operation. Lenient cons operation is decomposed into three operations, getcell, writecar and writecdr, each of which is executed in the Ref opn, car opn, and Cdr opn, respectively. The AN is designed so as to distribute getcell operations uniformly among SM banks.

TABLE I—Primitive functions in SM

OPN	FUNCTION	OPERATION
REF	INCREMENT	increments a reference count
	DECREMENT	decrements a reference count
	GETCELL	gets a new cell
ATTR	ATOM	tests for atomic cell
	EQ	tests for equality on atomic symbols
	NULL	tests for emptiness
CAR	WRITECAR	writes a value in the car field
	CAR	reads a value from the car field
	(CAR-G)	decrements a reference count of the cell pointed by the car field
CDR	WRITECDR	writes a value in the cdr field
	CDR	reads a value from the cdr field
	(CDR-G)	decrements a reference count of the cell pointed by the cdr field

How an operation car(x) is performed is illustrated by an example. The car opn takes an instruction packet from the instruction queue in AN interface and examines attribute information in the instruction packet. If the attribute data indicate that the cell is an atom, the error state is set into the result packet. Otherwise, the memory cell specified by the val field in the instruction packet is read from the ref block and the ready tag is checked. If the ready tag is on, a value z, which is read from the car field of the cell x, is returned to the IM as a result value. If the tag is off (which means the value has not yet arrived), the instruction packet is taken back to the tail of the instruction queue.

The garbage collection mechanism in Ref opn, which utilizes reference counter field, garbage tag field and garbage cell address buffer, is illustrated in Figure 9. The reference number is set to 1 when a getcell operation is executed and explicitly updated by increment or decrement operation.

When the reference count for a cell x becomes zero as a result of the decrement operation, the garbage cell address buffer is checked. If it is not full, the address of the cell x is stored in the buffer. Otherwise, a tag is set at the corresponding address in the garbage tag field. When room is made in the garbage cell address buffer by performing a getcell operation, the garbage tag field is searched and the address of the cell whose tag is set is stored in the buffer. Read and write accesses to the garbage cell address buffer are performed concurrently.

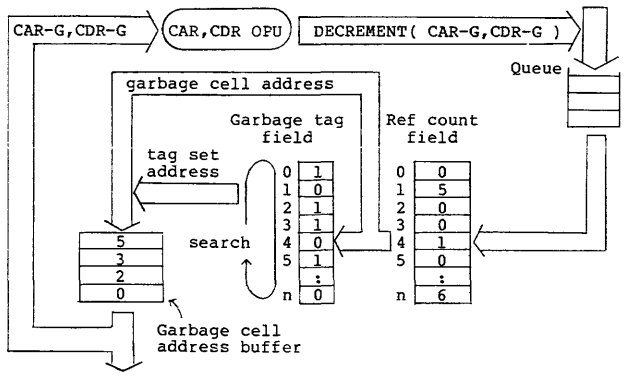


Figure 9—Garbage collection mechanism

The garbage tag search operation is interleaved with tag set operation; it does not itself set a tag. By using the garbage cell address buffer, a free cell address can be quickly obtained in the getcell operation.

6. CONCLUSION

This paper has presented an analysis of some issues concerning list processing under a data flow control environment from the viewpoint of parallelism and has also presented a new type of list-processing-oriented data flow machine, based on an associative memory and logic-in-memory.

The mechanism of partial execution in each function has been shown by example to be effective in exploiting the parallelism in list processing. The lenient cons mechanism has been shown to exploit maximally parallelism among activated functions.

In the list processing under data flow control, memory contention and garbage collection are serious problems. The problem of memory contention can be solved by dividing the structure memory into many banks and by uniformly distributing access in each bank, since new cells may be taken out from any bank.

The reference count is effective as a garbage collection method under a data flow control environment. The garbage collection algorithm presented here works well in the sense that cells are reclaimed whenever they are useless, concurrently with the foreground list operations.

Many problems remain to be solved for the data flow machine to be available for practical use. Several works are in progress to examine the effectiveness of the machine presented here. These include software simulator construction, experimental hardware system design, and VALID compiler implementation.

The simulator, which collects statistical information concerning the lenient cons effect, cons strategy and memory partition effect, and garbage collection overhead, etc., is now running. The experimental hardware system to estimate the cost performance is under development. The VALID compiler written in MacLISP is now under development on the DEC System 20.

ACKNOWLEDGMENTS

The authors wish to thank Dr. N. Kuroyanagi, the director of basic research division, and Mr. K. Yamashita, the director of first research section, for their continuing support and encouragement. They also wish to thank the architecture research group members in the first research section for fruitful discussions.

REFERENCES

1. Dennis, J. B. "A Preliminary Architecture for a Basic Data Flow Processor." *The Second Annual Symposium on Computer Architecture,* Jan., 1975, pp. 126–132.
2. Plas, A. "LAU System Architecture: A Parallel Data-Driven Processor Based on Single Assignment." *Proceedings of the International Conference on Parallel Processing,* 1976, pp. 293–302.
3. Watson, I., and J. Gurd. "A Prototype Data Flow Computer with Token Labelling." AFIPS, *Proceedings of the National Computer Conference* (Vol. 48), 1979, pp. 623–628.
4. Arvind, K., P. Gostelow, and W. Plouffe. "An Asynchronous Programming Language and Computing Machine." Report TR 114a, Department of Information and Computer Science, University of California, Irvine, California, December 1978.
5. Davis, A. L. "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine." *Proceedings of the Fifth Annual Symposium of Computer Architecture,* April 1978, pp. 210–215.
6. Keller, R. M., G. Lindstrom, and S. Patil. "An Architecture for a Loosely-Coupled Parallel Processor." UUCS–78–105, University of Utah, Salt Lake City, Utah, 1978.
7. Dijkstra, E. W. "Guarded Commands, Non-determinacy, and Formal Derivation of Programs." *Communications of the ACM,* 18 (1975), pp. 453–457.
8. Amamiya, M. "A Design Philosophy of High Level Language VALID for a Data Flow Machine." *Proceedings of IECEJ Annual Conference,* 1981, NO. 1486. In Japanese.
9. Friedman, D. P., and D. S. Wise. "CONS Should Not Evaluate Its Arguments." S. Michaelson and R. Milner (eds.), *Automata, Language and Programming,* Edinburgh: Edinburgh University Press, 1976.
10. Dennis, J. B., and K. S. Weng. "An Abstract Implementation for Concurrent Computation with Streams." *Proceedings of International Conference on Parallel Processing,* 1979, pp. 35–45.
11. Amamiya, M., R. Hasegawa, and H. Mikami. "A List Processing Oriented Data Flow Machine and Its Software Simulator." *Proceedings of Meeting on Computer Architecture,* IPSJ, 40–8, 1981. In Japanese.