ICOT Technical Report: TR-065

TR-065

Efficient Stream/Array Processing in Logic Programming Language

by

Kazunori Ueda (NEC)

and

Takashi Chikayama

April, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Efficient Stream/Array Processing in Logic Programming Language

by
Kazunori Ueda
(C&C Systems Research Laboratories, NEC Corporation)
and
Takashi Chikayama
(Institute for New Generation Computer Technology)

ABSTRACT

The Concurrent Prolog predicate for merging n input streams is investigated, and a compilation technique for getting its efficient code is presented. Using the technique, messages are transferred with the delay independent of n. Furthermore, it is shown that the addition and the removal of an input stream can be done in an average time of O(1). The predicate for distributing messages on an input stream to n output streams can also be realized as efficiently as n-ary merge. Mutable arrays that allow constant-time accessing and updating are realizable by the same implementation technique as that for distribute processes. Although the efficiency stated above could be achieved by a sophisticated compiler, the codes should be provided directly by the system to get rid of the bulk of source programs and the time required to compile them.

1. INTRODUCTION

In attempting to describe a large-scale distributed system in a parallel logic programming language such as Concurrent Prolog [Shapiro 83-1], the performance of the system will be influenced significantly by whether or not streams as interprocess communication channels can be merged and distributed efficiently. This paper deals with techniques for efficiently implementing predicates which merge many input streams and those which distribute data on a single input stream into multiple output streams.

This paper focuses on implementation techniques on conventional sequential computers. Of course, in order to demonstrate the practicality and viability of Concurrent Prolog on parallel computers, it is inadequate to limit the discussion to sequential computers. However, even after parallelism is implemented, it would be very likely for each processor to deal with multiple processes. In that event, the techniques presented here would be directly applicable to communication within one processor.

1.1. The Importance of Streams in Concurrent Prolog

Parallelism or coroutining in Concurrent Prolog can be realized by expressing individual processes via predicates which are executed in AND-parallel, and by enabling interprocess communication through shared variables. In the case, the shared variables express sequences of data or messages flowing among (usually two) predicates, and are normally represented as lists. As program execution proceeds, the values of the lists are gradually instantiated to the end. On the other hand, a predicate is the specification of (the relationship between) values that the shared variable as its arguments can take; procedurally, it can be regarded as a process which processes the sequences of data represented by shared variables from the top down (often through tail recursion). We use the term "stream" to refer to shared variables which are utilized in this manner.

Note that "process" and "stream" are pragmatic concepts of Concurrent Prolog, and are not grammatical concepts.

As is clear from the above explanation, communication with other processes is accomplished not by specifying the names of the processes, but by instantiating (in the case of sending) or by checking (in the case of receiving) the streams which have already been laid between processes. Therefore, the efficiency of stream operations--sending, receiving, merging, and distributing--are of crucial importance.

1.2. The Necessity of Dynamic, Multiway Stream Merging and Distribution

The merging and distribution of streams is unnecessary when several processes are linearly connected by shared variables to perform pipeline processing. However, if there is a process that needs to receive data or messages from many other processes--e.g. a process that manage shared resources--it is necessary to put the merging process at the front-end:

```
:- p1(C1), p2(C2), ..., pn(Cn),
   merge(C, C1, C2, ..., Cn), shared_resource(C).
```

In order to accept messages from an indefinite number of processes, it must also be possible to dynamically vary the number of input streams to be merged. In other words, if a process needs to communicate with shared processes, it is necessary to issue a request to the front-end merging process (by using other input streams or a 'request' stream), and to set up a new input stream. An alternative method to lay a new stream would be to attach a binary merge to one of the current input streams, but a delay proportional to the

number of communicating processes will arise if this method is repeatedly used.

As for message distribution, if it is done as "broadcasting", then each process need only share the broadcast stream. However, if it is necessary to communicate with a particular process but no stream exists leading directly to the process, communication must be enabled via the manager process of the destination process. The manager process must appropriately distribute messages according to the destinations attached to the messages.

Again, it is necessary to consider methods for dynamically varying the number of processes to be managed.

1.3. Previous Research

Shapiro et al. in [Shapiro 84] deals with the problem of merging indefinite multiple streams (henceforth the number of input streams will be denoted by n). They demonstrated

(1) A Method to ensure n-bounded waiting and a maximum delay of O(n) using an unbalanced tree consisting of binary mergers.
(2) A method to ensure n-bounded waiting and a maximum delay of O(log n) using a 2-3 tree [Aho 74] consisting of binary and ternary mergers.

The term "n-bounded waiting" was defined by them to mean that any message arriving at the merging process will be overtaken by no more than n input messages from other streams.

The delay of O(n) in Method (1) above is probably unacceptable when n is large enough and the traffic is heavy. This method may be practical, however, in the case of essentially costly communication such as interprocess communication in multi-processor environments.

Method (2) is a major improvement over (1) in terms of delay. In procedural languages, however, the delay of interprocess communication does not depend on the number of senders as long as it is simulated on a sequential computer. Therefore, also in logic programming languages, it is desirable to achieve a delay of the same order.

Gelernter in [Gelernter 84] discusses the suitability of Concurrent Prolog for describing multi-process systems. In his paper, he concludes that interprocess communication using merge networks are "not only bulky but unduly constricting". It should be noted, however, that this criticism is not from the viewpoint of descriptive power or efficiency.

2. OBJECTIVES

We have the following two objectives.

(1) When the number of input streams n is fixed, to realize on a sequential computer an n-ary merge with a maximum delay of O(1), and to realize an n-ary distributor with a maximum delay of O(1).
(2) To extend the solution to (1) to the case n varies dynamically.

In order to accomplish (1), it is clear that a successful result will not be obtained through the combination of binary and/or ternary merges. In other words, we must base the solution on n-ary merge as follows:

```
merge([X|Ys], X1, ..., Xj-1, [X|Xj], Xj+1, ..., Xn)
:- merge(Ys, X1, ..., Xj-1, Xj?, Xj+1, ..., Xn).
```

If this predicate is executed by an interpreter, the time required to process one message is proportional to the size of each clause (=O(n)), and is even less efficient than the balanced-tree method. This predicate, however, promises to yield higher efficiency if it is compiled, as will be discussed in 3.1. Thus, compiling techniques will be the main topic in this paper.

When utilizing n-ary merges, we cannot define "delay" as the depth of a merge tree. We will define the word "delay" as follows:

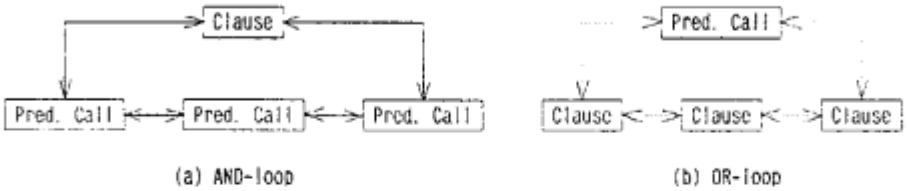
- The time passed from the arrival of a message at the merging process in an input-wait state until the original wait-state is restored by tail-recursion, during which an appropriate clause for processing the message is selected and the message is transferred to output streams. The delay is calculated by the number of primitive operations which can be accomplished in a unit time on a sequential computer.

Stream distribution will be dealt with in a manner similar to merge.

2.1. Outline of Sequential Implementation of Concurrent Prolog

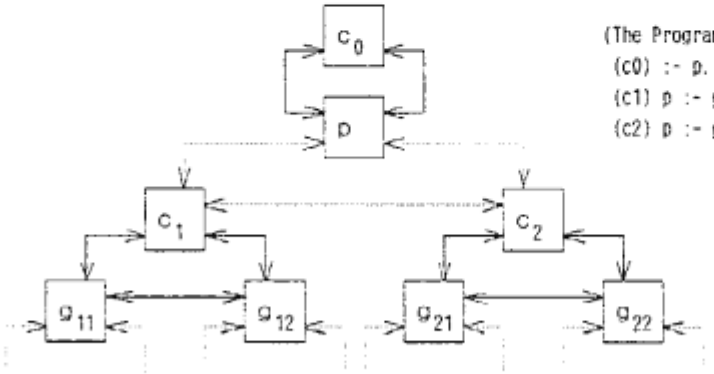
Examples of Concurrent Prolog implementation on a sequential computer include [Shapiro 83-1] and [Nitta 84], but both are interpreters. Here, we assume the implementation of a compiler which follows the guidelines stated in [Shapiro 83-2]. What follows is a brief explanation of the process management technique.

The descriptors of processes in an AND-relation (corresponding to a sequence of predicate calls) make up a circular list called an AND-loop, and the descriptors of processes in an OR-relation (corresponding to each clause composing a predicate) make up a circular list called an OR-loop (Figure 1-1,



(a) AND-loop (b) OR-loop

Fig. 1-1 AND-loop and OR-loop



(The Program)
(c0) :- p.
(c1) p :- g11, g12 | b11, b12.
(c2) p :- g21, g22 | true.

Fig. 1-2 Tree Structure Constructed by AND/OR-loop

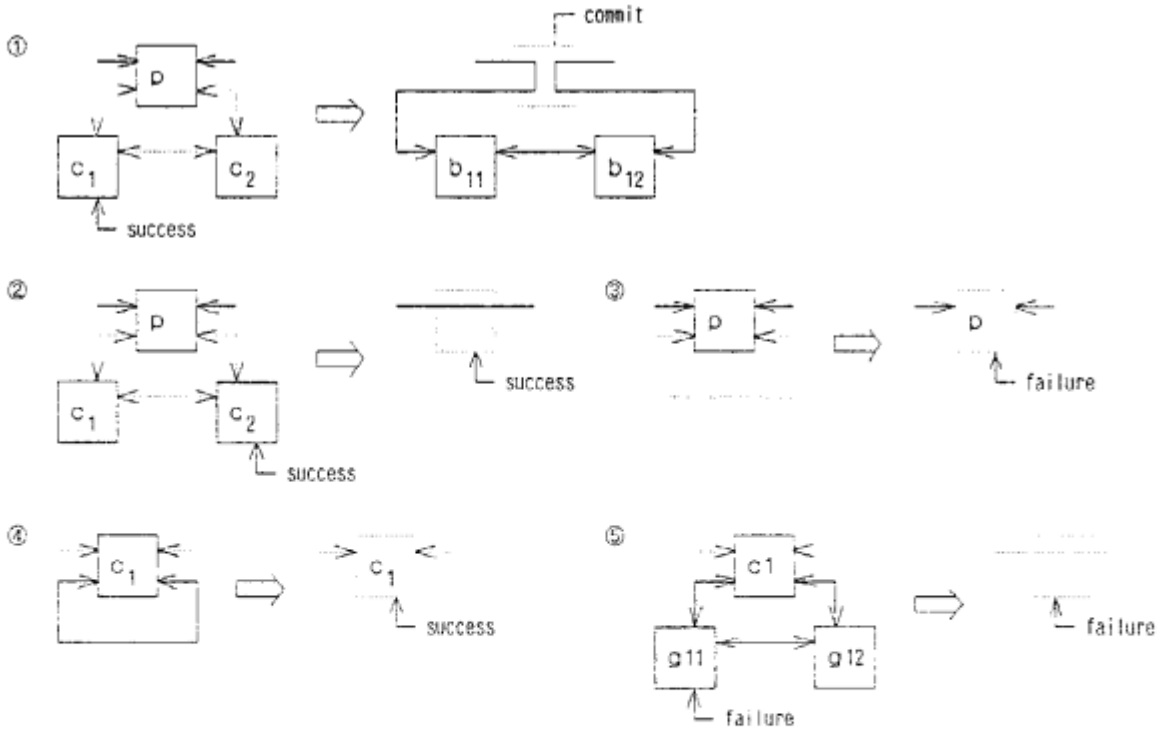


Fig. 1-3 Changes and Their Propagation of AND/OR loop

1-2).

An element of an AND-loop is, until it is "committed", the parent of an OR-loop comprising candidate clauses; after being committed, it is replaced by a doubly-linked list corresponding to predicate calls of the body. If the body is empty, then the element of the original AND-loop disappears. The parent of an AND-loop, having lost all elements, is considered a success. On the contrary, failure of any AND-loop element is regarded as a failure of the parent (Figure 1-3).

An element of an OR-loop represents a candidate clause which has not been "committed", and is the parent of the AND-loop whose elements represent predicate calls of the guard. The success of an OR-loop element implies the commitment of the corresponding clause. On the contrary, when an element of the OR-loop fail, that element simply disappears. If an OR-loop has lost all its elements, it is regarded as a failure of the parent (Figure 1-3).

There is a process queue in a system, and the leaf elements of a tree composed of AND/OR-loops (i.e. the elements which are not parents of other loops) are lined up and await scheduling. In unification, suspended clauses due to read-only variables are added onto the waiting lists attached to the read-only variables, instead of waiting in the process queue. These clauses will be rescheduled when the read-only variables are instantiated.

One possible optimization of the above method is to try to avoid creating OR-loops. This involves performing the unification of the clause head and the check of the simple guard as an indivisible operation (without making copies of writable variables). We call this "immediate check". If it is possible to commit with just an immediate check, we can avoid creating an OR-loop. In other cases, an OR-loop is created for those clauses which suspended during immediate check and those which have succeeded in the immediate check but which have complex guards, and they go into a wait state.

3. IMPLEMENTATION OF MERGE-PROCESSES

3.1. Examination of an n-ary Merge Predicate

N-ary merge can be expressed by n clauses of the following form, if one ignores the "base cases" which will be dealt with in Section 3.4.

```
merge([X|Ys], X1, ..., [X|Xk], ..., Xn)
:- merge(Ys, X1, ..., Xk?, ..., Xn).
```

This predicate has the following characteristics.

1. To see if the c-th clause is selectable, one need only test the unification

of the 0th and the c-th arguments (henceforth we number the arguments starting with 0).

2. In the case of tail recursion employing the c-th clause, only the 0th and the c-th arguments change compared with the original call. Therefore, if there is an argument list of the original call, it can be utilized to make a new argument list.
3. When all clauses are in a wait state, there is only one clause (two, even including the base case) which needs to be reexamined when one of the read-only variables as arguments is instantiated.

Now we will consider tail recursion. The arguments which do not change by tail recursion have the property that they do not alter the wait condition of each of the clauses. Suppose that a predicate is called, that the c-th clause is not selected due to the suspension (or failure) of the unification of the k-th argument, and that the d-th clause is selected instead. In this case, even after the tail recursion, the unification of the k-th argument of the c-th clause will be suspended (or will fail) if:

(1) the k-th argument of the c-th clause does not change by tail recursion, and
(2) the read-only variable that caused the suspension of the unification of the k-th argument of the c-th clause does not become instantiated by the unification of other arguments in the d-th clause.

If we state this in terms of the n-ary merge, we get the following.

4. In case of tail-recursion employing the c-th clause, the candidates are as follows:
   (a) the c-th clause itself
   (b) clauses which were candidates in a previous call but have not been examined.
   (c) clauses whose suspension have been terminated as the result of the instantiation of read-only variables.

   Possibility (c) does not exist under normal circumstances, so we can ignore it. Possibility (b) refers to the clauses that have been "carried over", so that once they are examined, they will either no longer be candidates (i.e. they will suspend or fail) or they will be selected and again become candidates after tail recursion. Therefore, the average number of clauses to be checked for each tail recursion does not depend on the number of candidates.

From the above considerations, we can conclude that the following operations which an n-ary merge undertakes for the processing of each message can be performed within a constant time.

(a) checking (= unification of the heads) of the candidate clauses.

(b) renewal of the argument list and suspension information accompanying tail recursion.

3.2. Implementation Technique for Merge of a Fixed Number of Inputs

To efficiently implement n-ary merge, the following are necessary.

(1) Even if all clauses suspend, an OR-loop (having O(n) elements) is not created, and they are made to wait at the predicate level.

(2) The argument list is re-utilized.

(3) In order to prevent the examination of clauses not worth examining, candidate clauses are managed within the process descriptor.

The configuration of the process descriptors and the implementation techniques of predicates following these guidelines are shown below. The methods shown below are applicable to predicates other than "merge", as long as they have no guards. In the following, the number of clauses composing the predicate will be denoted by M, and the number of arguments by N.

(I) Configuration of Process Descriptors

A process descriptor has the following items.

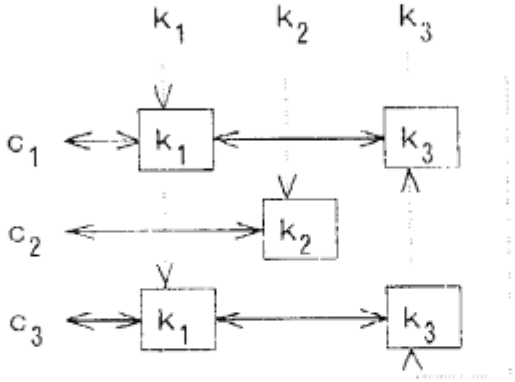


Fig. 2 Example of the Data Structure of Suspend/Fail Table

(1) AND Brothers: Two pointers for constructing the AND-loop.

(2) Backward Pointers: An array of pointers designating an entry on Process

Queue as well as entries on the waiting lists of read-only variables. The former is required one for each predicate, the latter one for each clause, so the total number of elements is M+1.

(3) Candidate Queue: The queue of candidate clauses of the call managed by the current process descriptor. M elements.

(4) Clause States: An array indicating whether each clause is in the "candidate", "suspend", or "fail" state. M elements.

(5) Suspend/Fail Table: The reasons why a particular clause was not selected can be attributed to some of the arguments of the caller. Thus, if these arguments change upon tail recursion, the clause may become selectable. Therefore, a table of pairs (c, k), where c is the number of the suspended or failing clause and k is the number of the argument that may be the cause, is maintained. This table must enable sequential retrieval with c as the index, as well as deletion of elements containing k, in the time proportional only to the number of elements to be retrieved or deleted. For example, the structure shown in Figure 2 fulfills this condition. The maximum number of elements depends on the program; in the case of merge, it is O(N+M)=O(n).

(6) Fail Count: The total number of clauses that cannot be selected for the current call. If this becomes M, the call fails.

(7) A pointer to the predicate's program code.

(8) Argument List: N elements.

(II) Operations

A. Creation of a Process Descriptor

When a predicate is newly called (that is, not as a tail recursion), the area for the process descriptor is allocated and each entry is set up. Here Backward Pointer (2) for process is made to point to the entry in Process Queue, and all Backward Pointers for clauses are set to "NIL". Furthermore,

* all clauses are entered in Candidate Queue (3),
* all Clauses States (4) are set to "candidate",
* Suspend/Fail Table (5) is cleared, and
* Fail Count (6) is set to 0.

B. Selection of a Clause

B-1. If Candidate Queue is not empty, instructions for unifying the head of the first candidate (say the c-th clause) and the arguments of the caller (Argument List of the process descriptor) are executed. For example, in the case of "merge", only the instructions for unifying the 0th and the k-th arguments are executed.

o If this succeeds, the body is executed (see D).

o If this fails,

(1) the binding generated is undone,

(2) Fail Count is incremented by 1,

(3) Clause State of the clause is set to "fail",

(4) Suspend/Fail Table is updated (cf. (III)), and

(5) other candidate clauses are tested.

o If this is suspended,

(1) the binding generated is undone,

(2) Clause State of the clause is set to "suspend",

(3) Suspend/Fail Table is updated,

(4) the pair (p, c), where p is the pointer to the process descriptor and c is the number of the clause, is entered in the waiting list of the read-only variable that caused the suspension,

(5) Backward Pointer for the clause is made to point to the pair entered in (4), and

(6) other candidate clauses are tested.

B-2. If Candidate Queue is empty and Fail Count is M (= the number of clauses), failure processing of the process takes place. Otherwise, execution of the current process is suspended.

C. Instantiation of Read-Only Variables

When a read-only variable is instantiated, the following is done for each entry (p, c) in its waiting list.

(1) The following is done for the process descriptor designated by p.

o Clause State of c is set to "candidate", and c is entered in Candidate Queue.

o All elements of the form (c, -) ("-" means "don't care") are deleted from Suspend/Fail Table.

(2) p is entered in Process Queue, and Backward Pointer for process is made to point to that entry.

D. Execution of the Body

If a recursive call is contained in the body of the committed clause (say the c-th clause), the following tasks are done.

(1) Assume that the arguments of the head and the arguments of the recursive call differ in the k1, k2, ..., kl-th arguments. For each ki (i=1, ..., l), the following are done.

- o Elements of the form (c, ki) are searched from Suspend/Fail Table, and for each c, the following are done.
  - o If Clause State of c is "fail", Fail Count is decremented by 1. If it is "suspend", the entry of the waiting list pointed to by Backward Pointer is eliminated.
  - o Clause State of c is set to "candidate", and c is entered in Candidate Queue.
  - o Elements of the form (c, -) are deleted from Suspend/Fail Table.
- o The ki-th element of Argument List is rewritten.

(2) The c-th clause is entered in Candidate Queue.

(3) Clause selection (cf. B) takes place.

If calls other than a recursive call are contained, new process descriptors are generated for them.

If there is no recursive call, the original process descriptor becomes unnecessary. Thus, the area is released after the pointers from the waiting lists of read-only variables are eliminated. However, there are cases in which this area can be used for optimization (cf. 3.4).

(III) Management of Suspend/Fail Table

If the c-th clause of n-ary merge is called as follows,

```
          0th      c-th
:- merge(Ys, ..., Xs?, ...).
```

unification of the c-th argument is suspended. In this case, the cause of suspension lies only in the c-th argument of the caller; even if another clause were selected and tail recursion took place, this would not remove the cause. However, we cannot always attribute the suspension/failure of the unification of the c-the argument only to that argument. For example, suppose the following example.

```
            0th     1st          c-th
:- merge([3|Ys], [3|Zs], ..., [2|Xs], ...).
```

If unification is done from the left, unification of the c-th argument fails, but we should attribute the cause also to the 0th argument. Actually, if the first clause is selected and tail recursion takes place, the c-th clause immediately becomes selectable.

To generalize, in case the unification of the k-th argument of the c-th

clause is suspended or fails, all arguments (numbered k1, ..., ki, ..., kl) "related to" the k-th argument in the c-th clause should be entered in Suspend/Fail Table in the form (c, ki).

Here, the term A is "related to" (henceforth denoted by R) the term B means that there are variables within A which are "related to" variables within B; and the variable V1 is related to V2 means that V1 and V2 are related by the reflexitive transitive closure of the following relation S.

Relation S: both variables appear together in a predicate call of the guard (if the guard is empty, S is the sameness of the variables).

Example: For the c-th clause of n-ary merge, the quotient set A/R of the set of arguments A by R is

{{0, c}, {1}, ..., {c-1}, {c+1}, ..., {n}}.

For the clause

p(I,J,K,L,M) :- a(I,J), b(J,K), c(L,M) | true.

we get

{{0,1,2}, {3,4}}.

However, it is necessary to make exceptions for the rules for updating Suspend/Fail Table. This is because if (0, c) is entered in Suspend/Fail Table when the c-th clause is suspended in a normal usage, this clause will be returned to Candidate Queue even by the tail recursion of another clause, and we cannot achieve the efficiency we desire. Therefore, in cases of suspension where

(1) the k-th argument of the caller is a read-only variable (viewed at execution time) and
(2) the k-th argument of the head is a non-variable term (viewed at compile time),

only (c, k) should be entered in Suspend/Fail Table. This is because it is clear that the cause of suspension is not in the other arguments related to the k-th argument.

The number of elements that are simultaneously entered in Suspend/Fail Table does not exceed

$$\sum \text{(the maximum size of the elements of "(set of arguments)/R")}$$

(the sum is taken with respect to clauses).

In the case of n-ary merge, this value is O(n).

3.3. The Properties of the Merge Predicate for a Fixed Number of Inputs

We will now examine the properties of n-ary merge when the compiling technique presented in 3.2 is employed. The existence of "base case" clauses will not be considered here. It will be discussed later in 3.4.

(I) Space Efficiency

The size of each process descriptor is O(n). The size of each item other than Suspend/Fail Table is clearly no greater than O(n), and the size of Suspend/Fail Table, as indicated in 3.2(III), is O(n).

The size of the program code will be discussed in (IV).

(II) Time Efficiency

A. The generation of process descriptors: O(n), but this need only be done once at the beginning.
B. Unification: The time required for the unification of the head of each clause is O(1), because there are two arguments on which unification should be attempted and the time for each does not depend on n. If a data structure such as the one shown in Figure 2 is assumed, the time required for the tasks accompanying suspended/failed unification (updating Suspend/Fail Table and the waiting lists for read-only variables) is also O(1).
C. The time for the instantiation of a read-only variable: O(1) for each task.
D. Tail recursion: The arguments that change when the c-th clause is selected is the 0th and the c-th arguments. However, as long as "merge" is used in the usual manner, the 0th argument will not be the cause of waits or failures, and the only clause waiting at the c-th argument is the c-th clause itself. Consequently, the only new candidate is the c-th clause. Furthermore, only two entries of Argument List need be rewritten. Therefore, the overall time required is O(1).

The above shows that the time required for processing a message reaching the n-ary merge in input-wait state does not depend on n.

(III) Order of Clause Checking

Individual clauses of n-ary merge are checked in the order they are entered in Candidate Queue. Since clauses which have been selected once are reentered at the tail of the list, n-bounded waiting is achieved. Moreover, as already stated in 3.1, suspended or failing clauses are removed from the list until the causes disappear; thus they do not influence the efficiency of the process.

(IV) Size of Program

The code for each clause describes operations B and D indicated in 3.2(II), so its size is O(1). Moreover, for operation A, only one piece of code is necessary per predicate, and its size is O(1). Consequently, the size of the code for n-ary merge is O(n).

However, it is possible to drastically reduce the code size. The codes for individual clauses are almost the same, so they can be parameterized with respect to the clause number c. If this is done, the code size for the whole predicate is reduced to O(1).

This parameterization could be accomplished by a sophisticated compiler capable of detecting similarities among the clauses. However, even if such a compiler were employed, it would not reduce the size of the source program (O(n**2)) and the time required for compilation. Furthermore, there may be only a few programs which can benefit from this optimization. Considering all these things, the most realistic approach is to let the system provide the code for n-ary merge.

Now we have n-ary merge at a code size of O(1). This, however, is still unsatisfactory. The system should provide "merge" for every n. If these were to be provided individually, the amount of code would be O(nmax), nmax being the maximum value of n.

However, here again, drastic optimization is possible. Because the codes for n-ary merges are almost the same regardless of the value of n, they can be parameterized with respect to n. This being done, the amount of code for merging any number of inputs becomes O(1).

Note that if the function of these codes were expressed in the form of source programs, the size would be O(nmax**3). Therefore, it is mandatory that these codes be provided by the system.

3.4. Dynamic Change of the Number of Input Streams

A merge predicate for a fixed number of input streams is useful only when the number of inputs is statically known. We will now expand this to allow the addition of new streams and the removal of terminated streams. The program is shown below. This program have an additional (the (-1)th) argument for accepting requests of new input streams. What this argument represents is the stream of new streams.

o The k-th clause (transfer)

```
    merge(S, [X|Ys], X1, ..., [X|Xk], ..., Xn)
    :- merge(S, Ys, X1, ..., Xk?, ..., Xn).
```

o The 0th clause (addition)

```
    merge([Xnplus1|S], Ys, X1, ..., Xn)
    :- merge(S?, Ys, X1, ..., Xn, Xnplus1).
```

o The (-k)th clause (removal)

```
    merge(S, Ys, X1, ..., [], ..., Xnminus1, Xn)
    :- merge(S, Ys, X1, ..., Xn, ..., Xnminus1).
```

o Base Case

```
    merge([], []).
```

These clauses are not tail recursive, since what the the bodies call are (n+1)ary and (n-1)ary merges. However, if process descriptors for these merges can be successfully constructed by modifying the original one for n-ary merge, it will be much more efficient than to create ones from scratch.

In Concurrent Prolog, process descriptors cannot be managed by simple stack scheme; a general memory management technique such as the Buddy system [Knuth 68] must be employed.

Here we will assume the use of the Buddy system. Then the size of each partitioned area will be a power of two. Each process descriptor is created in one of these areas. When it is created, its fields must be placed so that the cost of relocation with the addition or removal of streams is minimal. That is, the arrangement must be determined according to the size of the area allocated. Then, even if the number of inputs increases or decreases, most of the existing information need not be moved as long as the same area can accommodate all the information for the new descriptor.

Here we will show the operations to be performed when the (-n)th to 0th clauses are selected and the process descriptor can be reused. When considering the reuse of process descriptors, "unused" must be added as one of the possible states that Clause State can take, and when the area for Clause State is allocated, the unutilized portion should be set to "unused".

A. When the 0th Clause is Selected and an New Stream is Added

(1) (Operations accompanying the addition of the +-(n+1)th clause) If Clause States of the (n+1)th and the -(n+1)th clauses are not "candidate", they are set to "candidate", and those clauses are entered in Candidate Queue.
(2) The 0th clause is entered in the Candidate Queue.
(3) The (-1)th argument of Argument List is updated.
(4) The program code is replaced (If the program is parameterized with respect to n, only the parameter value is replaced).

B. When the (-c)th Clause (c>0) is Selected and an Empty Stream is Removed

(1) (Operations accompanying the change of the c-th argument) Elements of the

form (c', c) are retrieved from Suspend/Fail Table (even if such an element were to exist, (c, c) would be the only one). For each c', the following is done.

- o If Clause State of c' is "fail", Fail Count is decremented by 1; if it is "suspend", the entry in the waiting list pointed to by Backward Pointer for the c'-th clause is deleted, and then this Backward Pointer itself is deleted.
- o Clause State of c' is set to "candidate", and c' is entered in Candidate Queue.
- o Elements of the form (c', -) (only (c, c) can exist) are deleted from Suspend/Fail Table.

(2) (Operations accompanying disappearance of the +-n-th clause)

- o If Clause State of the n-th clause is "fail", Fail Count is decremented by 1. The same is done for the (-n)th clause.
- o Elements of the form (+-n, -) are deleted from Suspend/Fail Table.
- o (Nothing is done with the +-n-th clause in Candidate Queue. When they are dequeued and checked, nothing is due other than to change their Clause States to "undefined".)

(3) The (-c)th clause is entered in Candidate Queue.

(4) The c-th argument of Argument List is updated.

(5) The program code is replaced.

It is clear that both A and B can be accomplished in a constant time.

In case where a new stream must be added but the area for the current process descriptor cannot accommodate the new one, it is necessary to allocate a new area of twice the size and to move to that area. On the contrary, if it becomes possible to express the process descriptor with half the size of the current area (by the repeated removal of streams), the space between each item should be packed and the unused area collected should be freed. These operations are indicated below.

A', Addition of Streams Entailing Moving to a New Area

(1) An area twice the size of the current process descriptor area is allocated.

(2) All items of the original process descriptor are copied.

(3) The entries designated by all Backward Pointers are made to point to the new area.

(4) The operations described above in A are done.

B', Deletion of Streams Entailing Compaction

(1) The operations described above in B is done.

(2) Candidate Queue is examined and the +-n-th clauses are deleted, if any.
(3) The original process descriptor are packed in the top half of the current area.
(4) The bottom half of the area is released.

We will now consider the time complexity of A' and B'. If the time needed for allocating and releasing areas is ignored, both A' and B' can be done in a time proportional to n. The time complexity of the allocation and release of an area using the Buddy system is

O(log(size of the whole area managed by the Buddy system)).

This value, however, is determined only by the execution environment of the program, and is independent of n. Therefore, if the execution environment is fixed, the time needed for A' and B' is O(n).

In order to add and remove streams in an average time of O(1), it is necessary to guarantee that the frequency of doing A' or B' is at most once every O(n) times. However, this is easily achieved by doing B' only when it becomes possible to represent the process descriptor with (for example) one-fourth of the current area.

4. GUIDELINES FOR THE IMPLEMENTATION OF DISTRIBUTION PROCESSES

For the implementation technique of a distribution process, only outlines will be presented here.

4.1. Distribution to a Fixed Number of Output Streams

The predicate "distribute" with n output streams is expressed by n+1 clauses of the following form:

o The k-th clause

```
distribute([(k,X)|Xs], Y1, ..., [X|Yk], ..., Yn)
:- distribute(Xs?, Y1, ..., Yk, ..., Yn).
```

o The 0th clause

```
distribute([], [], ..., []).
```

First, we will consider the situation where there is no wait. It is necessary to implement random accessing of clauses because, if the 1st to k-th clauses were individually checked, the time required would be O(n). The Dec-10 Prolog compiler [Warren 77] generates a code that selects clauses using the hash value of the principal functor of the first argument. However, this is

inadequate for stream-oriented programming. In the case of "distribute", hashing by the tertiary functor (a functor of the third level) of the first argument is necessary to select a clause in constant time.

Next, as we did with "merge", we will consider how to achieve the code size of O(1). It is of course necessary to parameterize the codes of each clause. In the case of "distribute", we should further make use of the fact that clauses can be selected by simple indexing which does not involve hashing: a hash table requires an area of O(n).

What if there is a wait? When this predicate is used in a usual manner, the 0th argument becomes the cause of the wait. However, if the 1st to k-th clauses all individually go into wait, the goal of efficiency cannot be achieved. The 1st to k-th clauses should always be managed together: not only when indexing, but also while "waiting". In other words, they should be entered in the waiting lists of read-only variables as a cluster of clauses. When their suspension are released, the appropriate clause should be selected by indexing.

4.2. Dynamic Change of the Number of Output Streams

As in the case of merge, a capability for dynamically changing the number of output streams is important. This can be implemented by adding the following clauses:

o Addition

```
distribute([grow(Ynplus1)|Xs], Y1, ..., Yn)
:- distribute(Xs?, Y1, ..., Yn, Ynplus1).
```

o Deletion

```
distribute([shrink|Xs], Y1, ..., Ynminus1, Yn)
:- distribute(Xs?, Y1, ..., Ynminus1).
```

In order to efficiently change the number of output streams, a method similar to the one described for "merge" in 3.4 should be applied.

5. APPLYING IMPLEMENTATION TECHNIQUE OF DISTRIBUTION PROCESSES TO MUTABLE ARRAYS

The lack of mutable arrays (arrays of rewritable elements) is often mentioned as one of the problems of Prolog. Of course, arrays can be simulated by using "assert" and "retract", but these are not "logical" arrays. One direction to realize logical arrays is to make a correspondence

arrays: data of the array type
operations on arrays: predicates having array arguments

and to gain efficiency by dedicated data structure. However, it is also possible to make the following correspondence

arrays: predicate calls (processes)
operations on arrays: messages in streams,

and to realize constant-time accessing and updating by using the same implementation technique as that of the distribution process. This is a rather natural solution if we regard arrays as mutable "objects". The program would be as follows:

```
array(n, S) :- array(S, X1, ..., Xn).
array([read(k,Xk)|S], X1, ..., Xk, ..., Xn)
:- array(S?, X1, ..., Xk, ..., Xn).  (for k=1, ..., n)
array([write(k,Yk)|S], X1, ..., Xk, ..., Xn)
:- array(S?, X1, ..., Yk, ..., Xn).  (for k=1, ..., n)
```

It is also possible to add clauses for inquiring and/or changing the number of elements.

6. CONCLUSIONS AND FUTURE WORKS

The properties of n-ary merge written in Concurrent Prolog were investigated and an implementation which transfers each message with the delay independent of n was shown. Furthermore, it was shown that an input stream can be added and removed in an average time of O(1). With respect to n-ary distribution also, outlines for an implementation as efficient as "merge" were presented. Mutable arrays that allow constant-time accessing and updating are shown to be realizable by the same implementation technique as that for distribution processes.

However, it was concluded that these predicates should be supplied by the system. If the system provides them, it is possible to realize merge and distribution for all n with the code whose size does not depend on the maximum number of n.

On the other hand, to obtain the code by compiling a source program provided by the user is unrealistic, not from the viewpoint of the efficiency of the code obtained, but from the viewpoint of the bulk of the source program and the time needed for compilation. However, it is favorable in many respects

(e.g. for the construction of programming systems) that the semantics of the system-supplied code is expressible as a Concurrent Prolog program.

The suggested technique for the implementation of n-ary merge has a problem that it does not work efficiently when a bounded buffer [Takeuchi 83] is connected to the output stream. However, it is expected that this problem can be solved by improving the method of clause wait and scheduling.

The most important future tasks are to describe large-scale systems in Concurrent Prolog, to estimate the cost of interprocess communication, and to confirm the usefulness of the suggested capabilities. It is also important to consider an efficient implementation of interprocess communication in parallel environments.

ACKNOWLEDGMENTS

The authors thank to Katsuya Hakozaki, Masahiro Yamamoto, Kazuhiro Fuchi, and Kouichi Furukawa for providing a stimulating place in which to work. Thanks are also due to Ehud Shapiro for offering them hints to embark on this study, as well as to Akikazu Takeuchi for valuable suggestions.

REFERENCES

[Aho 74] A. V. Aho, J. E. Hopcroft, J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, Reading, Mass. (1974).

[Gelernter 84] David Gelernter, "A Note on Systems Programming in Concurrent Prolog", Proc. 1984 Int. Symp. on Logic Programming, pp.76-82 (1984).

[Knuth 68] D. E. Knuth, The Art of Computer Programming, Vol.1: Fundamental Algorithms, Addison-Wesley, Reading, Mass (1968).

[Shapiro 83-1] E. Y. Shapiro, A Subset of Concurrent Prolog and Its Interpreter, ICOT Tech. Report TR-003, Institute for New Generation Computer Technology (1983).

[Shapiro 83-2] E. Y. Shapiro, Notes on Sequential Implementation of Concurrent Prolog: Summary of Discussions in ICOT (1983)(unpublished).

[Shapiro 84] E. Y. Shapiro, C. Mierowsky, "Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog", Proc. 1984 Int. Symp. on Logic Programming, pp.83-90 (1984).

[Warren 77] D. H. Warren, Implementing PROLOG--Compiling Predicate Logic Programs, Vol.1-2, D. A. I. Research Report No. 39, Dept. of Artificial Intelligence, University of Edinburgh (1977).

[Takeuchi 83] Akikazu Takeuchi, Kouichi Furukawa, "Implementing Interprocess Communication in Concurrent Prolog", 27th IPSJ National Conference, 3E-7 (1983) (in Japanese).

[Nitta 84] Katsumi Nitta, "On Concurrent Prolog Interpreter", to appear.