PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988



# LogDf : A DATA-DRIVEN ABSTRACT MACHINE MODEL FOR PARALLEL EXECUTION OF LOGIC PROGRAMS[1]

Prasenjit Biswas and Chien-Chao Tseng

Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275

## ABSTRACT

The abstract data-driven machine model, named LogDf, is developed for parallel execution of logic programs. The execution scheme supports OR-parallelism, Restricted-AND parallelism and stream parallelism. Multiple binding environments are represented using stream of streams structure (S-stream). Eager evaluation is performed by passing binding environment between subgoal literals as S-streams, which are formed using non-strict constructors. The hierarchical multi-level stream structure provides a logical framework for distributing the streams to enhance parallelism in production/consumption as well as control of parallelism. The scheme for compiling the dataflow graphs eliminates the necessity of any operand matching unit in the underlying dynamic dataflow architecture. The details of binding representation and efficient representation for structures/lists are also included.

## 1. INTRODUCTION

Two fundamental problems related to the overhead of synchronization and latency seem to be unavoidable in control driven multiprocessing [6]. Dataflow execution model provides an efficient alternative. At the abstract level dataflow execution model provides the possibility of exploiting maximal parallelism, at the finest level of granularity. In an implementation of the execution model using finite resources, the high degree of parallelism provide the necessary capability for tolerating memory latency and delays in the communication network. In essence, the abstract execution model (not necessarily the implementation) provides the framework for extracting maximal inherent parallelism, as well as provide the flexibility of grain size determination for an efficient implementation [23].

Dataflow execution model is purely functional in nature and thus has evolved as one of the primary execution models for functional or single assignment languages. Logic programs, though not functional in nature, have certain properties that have attracted researchers to develop data-driven models for parallel execution of logic programs [4,9,19,21,25,27]. The inherent parallelism in logic programs is naturally exploited by the dataflow execution mechanism.

Typically the clauses in a logic program are viewed as procedures. The variables used in a clause are local to the clause and the same variable name is treated as a different entity in another clause. This notion of *local scoping* particularly makes logic programs amenable to dataflow model of execution. The clauses in an non-annotated logic program (when viewed as procedures) differ from functions in that the input/output relationships of the arguments are decided at run time. Moreover, the procedure invocation during execution is non-deterministic. These characteristics make the dataflow execution schemes for parallel logic programs different from the execution schemes for purely functional counterpart.

In this paper we consider the logic program is non-annotated (i.e., without mode declarations, read-only annotations or any other control pragma). Moreover we consider it important to implement "don't know" non-determinism and assume the top level query may require all possible solutions.

### 1.1 Salient Features

Some of the salient features of the LogDf abstract machine model are summarized below and will be elaborated in the remaining sections of the paper.

(1) The execution model supports OR-parallelism, Restricted-AND parallelism and Stream parallelism in logic programs.

(2) Eager evaluation is supported by representing binding environment, produced by the solution of a subgoal, as a non-strict multilevel stream of streams structure (abbreviated as S-stream).

(3) S-streams allow high degree of parallelism in production and consumption of binding environments. The architecture provides efficient support for parallel decomposition of the S-streams.

(4) For efficient implementation of OR-parallelism, it is imperative to provide an efficient mechanism for storage and access of multiple binding environments [11,14,15,26]. The hierarchical S-stream structure

---
[1] This research was supported in part by Texas Advanced Technology Program Contract 3128 (1988).

provides a principle for distribution of the binding environments over multiple structure memories as the relationship between the producer process and the levels of the stream is explicitly represented in the structure. The structure also allows a fairly straightforward procedure for detecting the end of stream ('eos') condition, which is extremely important for conserving resources in a highly parallel environment.

(5) The binding environment associated with a cell of the S-stream is 'closed'[14] and thus no dereferencing to other levels is required. Moreover, the indexed binding representation provides constant time access to any variable binding in the frame.

(6) The execution mechanism is based on tagged token dynamic dataflow principle to support multiple activation of reentrant dataflow graphs and recursion. The principle of compilation in LogDf allows the elimination of the operand matching unit (a typical bottleneck in the implementation of the dynamic dataflow principle).

(7) The 'sync' problem [22] associated with restricted Cartesian product operation is naturally solved in this dynamic dataflow framework without incurring any additional overhead.

## 2. BACKGROUND

In the following discussion, we will view a clause as a procedure [20]. So a goal statement or the body of clause

$$:-A_1, A_2, ...$$

would be viewed as a set of procedure *calls* constituting the body of the procedure represented by the clause. The subset of clauses in the program with similar head literals (i.e., same predicate name and number of arguments) will be referred to as *candidate clauses* for a procedure call. An entry to a procedure would take place on successful unification of the goal literal (equivalent to procedure call) with the head literal of a clause representing the procedure.

A pure Horn clause logic program offers many possibilities for exploiting parallelism. The two most common forms of parallelism inherent in logic programs are OR parallelism and AND parallelism [13]. Many other forms of parallelism in execution have been reported which have evolved essentially from implementation dependent restrictions.

OR-parallel execution implies the execution of all the clauses whose head literals unify with the goal. AND parallel execution involves execution of all the body literals of a clause in parallel.

The major problems associated with AND/OR parallel execution are the issues pertaining to management of binding information and relating variable bindings to activations of these parallel processes. In this paper we propose an elegant solution to this problem in the context of data driven parallel execution of logic programs.

The proposed model has similarities with two models reported earlier [4,21]. The basic similarity lies with the concept of passing binding environments between goal literals in the form of a structure that is formed using non-strict constructors [1,2,3,17].

The model proposed by Amamiya and Hasegawa [4] is an interpreter for logic programs developed in Valid-E for dataflow machine DFM. The structure used in this model is a tree structure where binding information is at the leaf cells. The consumer process (the process to solve a goal literal) traverses the tree to identify the leaves before using them. Moreover, the tree structure might contain 'fail' cells scattered through out which are not detected until a process attempts to consume one.

In the other model proposed by Ito et.al. [21], the structure used is a stream, global to a number of OR-processes. Each of these processes may append sets of bindings to this global structure or number of processes may consume the binding environment, thereby causing the usual bottleneck problems. Due to the non hierarchical nature of the structure it is difficult to relate a set of bindings to process activations and establish any principle of distribution of the stream over multiple structure memories. Moreover, in a stream oriented processing model supporting OR parallel execution, it is extremely important to detect the end-of stream (eos) information efficiently. The counter scheme proposed in their report for determining 'eos' would become extremely complex as it would require propagating counter updating information through the levels of the proof tree. The grain of parallelism seems to be too fine to be useful when AND/OR parallel execution tends to be combinatorially explosive in number of processes.

## 3. THE PROPOSED DATA-DRIVEN MODEL

As indicated in the previous section, solving of a subgoal is viewed as procedure call, where the procedures correspond to the candidate clauses. A call corresponds to an instance of a subgoal, where the subgoal literal is instantiated with appropriate binding environment. The dataflow graph for solving an instance of a subgoal literal for a particular input binding environment (henceforth referred to as a *BE frame*[2]) is shown in figure 1 (SOLVE_LIT). The graph is activated on arrival of an input BE frame. The Activate node forms an instance of the subgoal literal expression using the input BE to produce a goal token (henceforth referred to as a *goal frame*). The Activate node is associated with a literal expression which is a constant argument of the node. The goal frame is distributed to the inputs of the dataflow graphs for the procedures corresponding to the candidate clauses for the subgoal. All the procedures are invoked in parallel, thus

[2] The BE frames at any level contains the current status of bindings for the variables. In the BE frame for the first subgoal in the top-level query, all the variables are unbound.

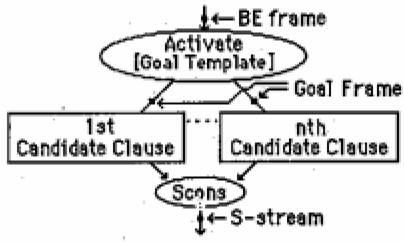


**Fig.1. SOLVE_LIT**

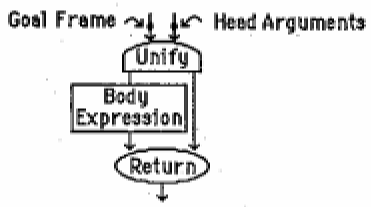


**Fig.2. Clause Representation**

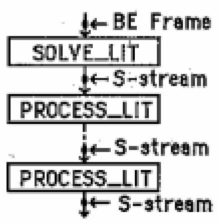


**Fig.3. Body Expression**

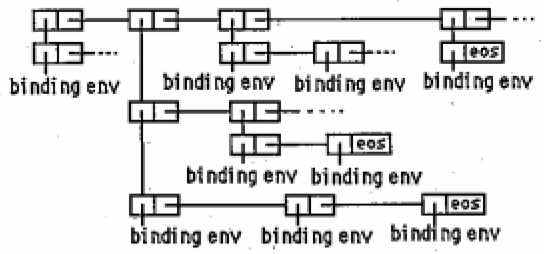


**Fig.4(a). S-stream Structure**


**Fig.4(b). Cell Format**

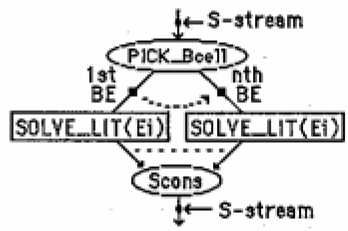


**Fig.5. PROCESS_LIT**

achieving OR-parallel execution.

The procedure graph corresponding to a clause is shown in figure 2. The Unify node corresponds to the clause head. The literal expression for the head of the clause is a constant argument associated with this node. The body-expression block shown in the figure represents the dataflow graph for the body of the clause. The dataflow graph for the body of the clause is a SOLVE_LIT graph connected to a sequence of PROCESS_LIT graphs as shown in figure 3. The SOLVE_LIT graph corresponds to the first subgoal literal in the body of the clause. These PROCESS_LIT graphs have 1-1 correspondence with the other subgoal literals in the body of the clause. As indicated in the figure, the BE frames obtained from the solution of a subgoal literal is passed on to the next subgoal in the body, in the form of a non-strict structure termed as an S-stream (stream of streams).

A typical S-stream is shown in figure 4(a). The principles underlying the construction of the structure is clearly explained in the next section. The structure consists of two kinds of cells- Bcell and Scell. A cell has two fields (car and cdr) as shown in fig. 4(b). The car field of a Bcell contains a pointer to BE frame (a list of variable bindings). The car field of an Scell contains a pointer to another S-stream. The cdr field of both types of cells contains either a pointer to the next cell in the same level of stream or an end of stream (eos) indicator. A cell in the stream is created only when the car field could be assigned a non-null value.

In the proposed model, we support eager evaluation, by forwarding the pointer to the beginning of an S-stream to the consumer function, whenever the first cell of the stream gets allocated. The cdr field of a cell has a ready (R) and a pending (P) bits associated with it. The R bit indicates whether the value in the field is defined or not. Any operator trying to read the cdr field gets suspended if R bit for the field is not set. The P bit indicates that there are pending requests for the value of this field. When the R bit gets set, the value is forwarded to the suspended operators.

A SOLVE_LIT($E_j$) graph represents a subgoal literal $E_j$. As shown in figure 3, it is sufficient to represent the first subgoal in the body of a clause by a SOLVE_LIT graph as only one BE frame is generated by a head unification. It should be obvious that each subsequent subgoal in the body of a clause will have to be solved for multiple BE's returned by the candidate clauses of the previous subgoal in the sequence. This feature of OR-parallel execution necessitates the representation of each subsequent subgoal literals ($E_i$) by higher order PROCESS_LIT($E_i$) graphs. The graph for PROCESS_LIT($E_i$) is shown in figure 5. The activating input of a PROCESS_LIT graph is an S-stream of BE's produced by the previous SOLVE_LIT or PROCESS_LIT graph in the sequence of subgraphs in the body expression. The PICK_Bcell operator collects BE's from the stream to provide one BE per activation for separate activations of the same SOLVE_LIT graph.

The only other node in the Clause graph is the Return operator. the Return operator uses the return environment information (produced by Unify) to extract the bindings of the output variables in the head of the clause, from the stream of BE's received by the node.

### 3.1 An Example (OR-parallel)

Now we will use an example logic program to complete the overview of the execution mechanism.

C1 : P(X,Y) :- Q(X,Z), R(Z,Y).
C2 : Q(a,b).
C3 : Q(a,c).
C4 : Q(b,c).
C5 : R(b,f).
C6 : R(c,g).
C7 : R(Z1,Y1) :- S(Z1,Y1).
C8 : S(c,i).
C9 : S(c,j).

:- P(a,XX).

The query literal is P(a,XX). There is only one candidate clause (C1) for solving the query subgoal. We show the clause representation graph of C1 in fig. 6(a). The goal literal is P(a,XX) and the head literal is P(X,Y). In the figure, we show the respective representations at the flow graph level. The goal literal is represented by the goal frame which contains the bindings of the goal arguments and also a pointer to the binding environment of the query. Similarly, the head literal is represented by the arguments (cf, section 4 on binding representation). The status of the BE corresponding to the body of the clause, after head unification is shown at the left output of the Unify node. The outputs Pt_Q, Pt_R and Pt_C1 shown in the figure are pointers to streams of BE's produced by the SOLVE_LIT, PROCESS_LIT blocks and the Return node respectively.

In fig. 6(b), we show the internals of the SOLVE_LIT block for the literal Q(X,Z). There are three alternative candidate clauses (C2, C3 and C4) for solving the subgoal Q(X,Z) and all the three are assertions. The unification of Q(a,Z) with Q(b,c) fails and the block returns a null BE. The BE's returned by the other two assertions are [X/a, Y/Φ, Z/b] and [X/a, Y/Φ, Z/c]. The Stream Cons operator allocates a cell for the new stream (pointed by Pt_Q) on receipt of a non_null input token. Once the first cell in the new stream is created the pointer (address Pt_Q) is forwarded to the consumer graph (in this case PROCESS_LIT graph for R(Z,Y)). When the stream Cons operator receives additional non_null input tokens, it creates new cells to hold the token value and appends them to the stream. The single level stream Pt_Q is shown in fig. 6(b).

The pointer Pt_Q of the stream is input to the graph for PROCESS_LIT R(Z,Y). As shown in fig. 7(a), the function of PICK_Bcell operator is to select the BE frame

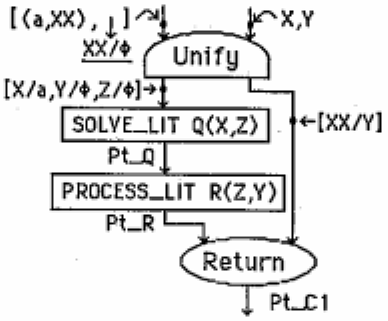


Fig. 6(a). Clause C1

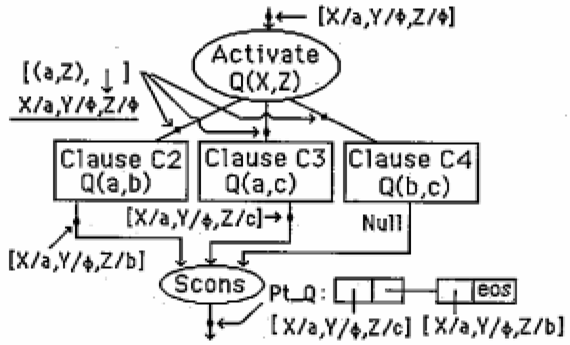


Fig. 6(b). SOLVE_LIT Q(X,Z)

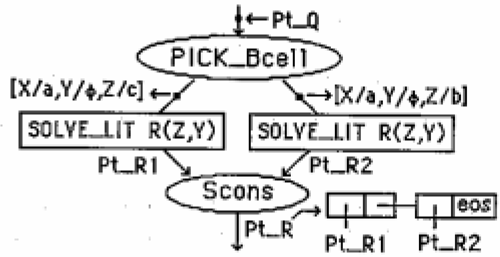


Fig.7(a). PROCESS_LIT R(Z,Y)

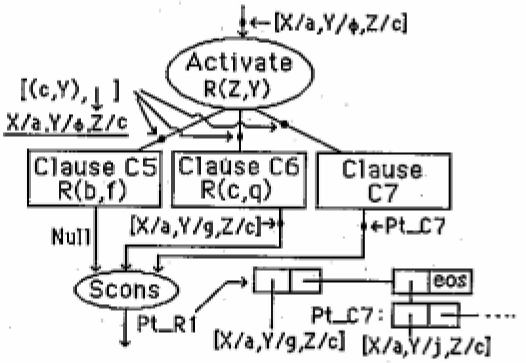


Fig.7(b). An Instance of SOLVE_LIT R(Z,Y)

pointers (held in the car field of a Bcell) from the input stream and cause two different activations of the

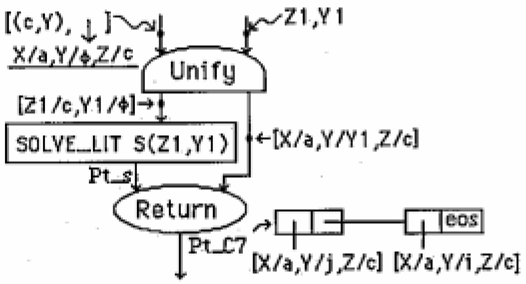


Fig.8(a). Clause C7

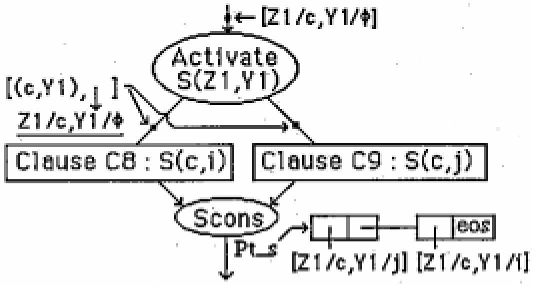


Fig.8(b). SOLVE_LIT S(Z1,Y1)

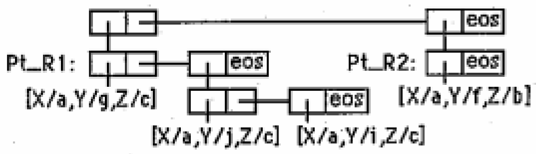


Fig.8(c). Stream Pt_R

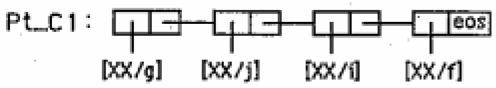


Fig.8(d). Stream Pt_C1

SOLVE_LIT R(Z,Y) graph. The outputs from these two activations of the SOLVE_LIT R(Z,Y) graph are shown as Pt_R1 and Pt_R2, the pointers to the stream of BE's produced by the two graphs. It should be noted that the pointer Pt_R is forwarded to the return operator (cf., fig. 6(a)) immediately after the first cell of the S-stream is allocated.

One of the instances of the SOLVE_LIT R(Z,Y) subgraph of PROCESS_LIT R(Z,Y) graph is shown in fig. 7(b). The principle of the construction of the stream Pt_R1 is similar to the stream Pt_Q discussed earlier. The second level stream of stream Pt_R1 (pointed by Pt_C7) is produced by clause C7 (as shown in fig. 8(a) and 8(b)).

As Pt_R1 points to a 2-level stream structure, the stream pointed by Pt_R (in fig. 7(a)) has 3 levels. The structure of the 3-level stream is shown in fig. 8(c). The return operator in fig. 6(a) decomposes the 3 level stream structure to extract the necessary binding information, (as prescribed in the Return Environment token) to produce the single level stream Pt_C1 as shown in fig. 8(d).

### 3.2 Extensions for Restricted-AND Parallelism

The extensions required for supporting Restricted-AND-Parallelism (RAP) [16] in the execution scheme will be briefly discussed here. Details were provided in [24]. The conditional graph expressions (CGE's) representing the logic program are compiled into dataflow graphs. To incorporate the different types of CGE's, the PROCESS_LIT graph is generalized into PROCESS_EXP graph as shown in figure 9, where SOLVE_EXP could be any of the five graphs shown in figures 10(a) through 10(d). The remaining descriptions in this paper are based on the combined OR/RAP model.

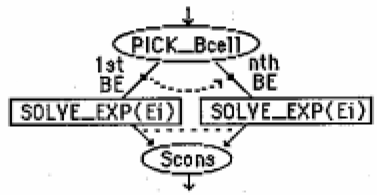


Fig.9. PROCESS_EXP

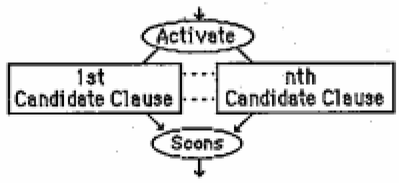


Fig.10(a). SOLVE_LIT

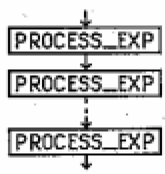


Fig.10(b). SOLVE_SEQ

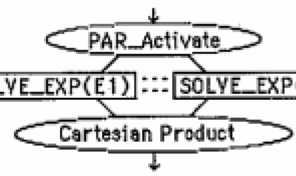


Fig.10(c). SOLVE_PAR

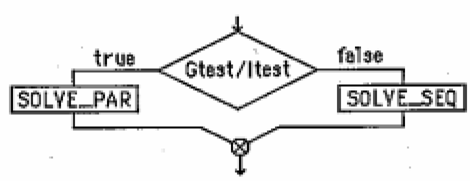


Fig.10(d). SOLVE_GTEST/SOLVE_ITEST

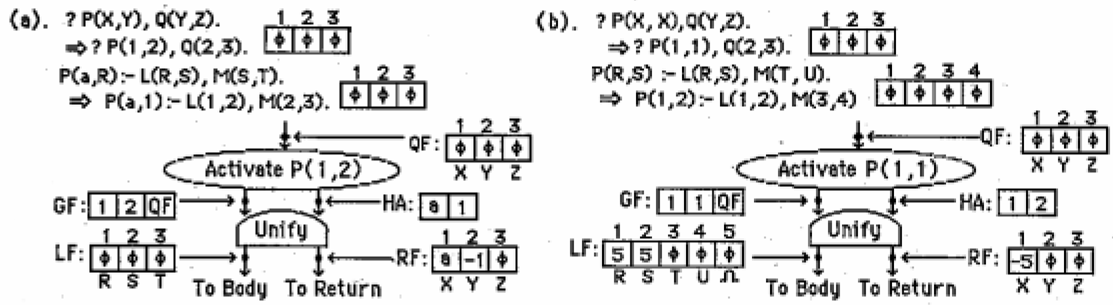


**Fig.11. Frame Representations and Relationships**

| HA \ GF | [1,1,QF] LF | [1,1,QF] RF | [1,b,QF] LF | [1,b,QF] RF | [2,3,QF] LF | [2,3,QF] RF |
|---|---|---|---|---|---|---|
| [1,2] | [4,4,Φ,Φ] | [-4,Q2,Q3] | [Φ,b,Φ] | [-1,Q2,Q3] | [Φ,Φ,Φ] | [Q1,-1,-2] |
| [1,1] | [Φ,Φ,Φ] | [-1,Q2,Q3] | [b,Φ,Φ] | [b,Q2,Q3] | [Φ,Φ,Φ] | [Q1,-1,-1] |
| [1,b] | [b,Φ,Φ] | [b,Q2,Q3] | [Φ,Φ,Φ] | [-1,Q2,Q3] | [Φ,Φ,Φ] | [Q1,-1,b] |
| [a,1] | [a,Φ,Φ] | [a,Q2,Q3] | [b,Φ,Φ] | [a,Q2,Q3] | [Φ,Φ,Φ] | [Q1,a,-1] |
| [a,b] | —— | —— | [Φ,Φ,Φ] | [a,Q2,Q3] | [Φ,Φ,Φ] | [Q1,a,b] |

Q1,Q2,Q3 : the 1st, 2nd and 3rd field value of QF respectively

**Fig.12. Local Frames and Return Frames Created by Unification**

## 4. REPRESENTATION OF BINDING ENVIRONMENT

The binding environment is represented in memory using a vectorized format, named as a variable frame (Vframe). The local variables of a clause are compiled into serial indices, where each index corresponds to a cell in the Vframe. For example, let us consider the clause -

P(a,B) :- Q(B,C), R(C,D).

The Vframe associated with this clause is -

| B | C | D |
|---|---|---|
| Φ | Φ | Φ |
| 1 | 2 | 3 |

,where Φ stands for unbound

The internal representation of the clause would be -

P(a,1) :- Q(1,2), R(2,3).

We classify the variable frames into three different categories :

(i) Query frame (Q-frame) : Vframe associated with the query or goal statement.

(ii) Local Frame (L-frame) : represents the binding of the variables of the clause body after successful head unification. The indices refer to cells in the same frame.

(iii) Return frame (R-frame) : represents the Return environment. The Return environment stands for the status of the variable bindings of the parent Q-frame after unification. A positive index refers to the same frame (i.e., R-frame). and a negative index refers to a cell in the L-frame.

The goal and the head literal are represented as follows :

(1) Goal frame (G-frame) : represents the binding of the variables of a goal. It is represented as an array of cells, where a cell corresponds to an argument in the goal literal. An index in a cell refers to the Q-frame, a pointer to which is part of the G-frame.

(2) Head argument (HA) : represents the bindings of the arguments of the head literal of the clause. The indices refer to the Vframe associated with the clause.

Figures 11(a) and 11(b) are provided to clarify the representations and relationships between the different types of frames. Figure 11(b) needs some explanation. The unification of the goal literal P(X,X) and the head P(R,S) causes R to be bound to S. The sharing of two unbound variables is represented by creating a fifth cell (Ω) in the LF. The negative index in the RF refers to this new cell Ω in LF.

We do not present the details of the unification algorithm in this paper. Figure 12 shows various forms of L-frame and R-frame that could be created by unification for different combinations of G-frames and head arguments.

### 4.1 Structure/List Handling

In parallel execution of logic programs, it is important to avoid copying of the whole structure to represent the different bindings for the uninstantiated variables in the original structure. For the ground structures (structures formed by ground terms), the question of multiple binding conflict does not arise and they can be easily shared. However, for non-ground structures, i.e. structures containing unbound variables, the variables may be instantiated to different bindings at different stages of execution. In this section, we will present a straightforward scheme to support structure sharing.

In this scheme, a structure is represented as an

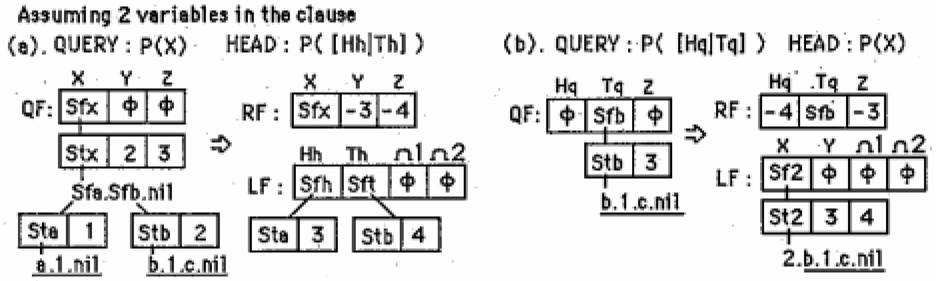


**Fig.13. Frame Relationships Before and After Unification**
(sharing of sublists is indicated by the portions underlined)

Sframe which contains a pointer to a structure template, Stemp, and a variable frame, SVframe. In the Stemp, each element is either a constant term, a variable or a pointer to a nested substructure (an Sframe). The SVframe, represented as a vectorized format, contains the bindings of the variables which appear in the Stemp. Each distinct variable in the Stemp is assigned a unique number and this number is used as an index to access the corresponding cell of the SVframe. For example, assuming the binding environment represented by Vframe-1 is—

| | W | X | Y |
|---|---|---|---|
| Vframe-1 : | Φ | Φ | Φ |
| | 1 | 2 | 3 |

,where Φ stands for unbound,

and we want to represent the structure g(X,Y). First of all, the variables X and Y in g(X,Y) are numbered 1 and 2 respectively, and the Stemp for the structure g(X,Y) is represented as g(1,2). The SVframe is

| 2 | 3 |
|---|---|

because the bindings of variables X and Y are respectively in the second and the third cells of Vframe-1. If pt1 is the pointer to the Stemp g(1,2) the structure g(X,Y) can be represented as

Sframe-g1 : | pt1 | 2 | 3 |

For nested structures (structures containing substructures) the same principle holds. However, the cells of the SVframe in a substructure contain indices to the cells of the immediate higher level SVframe instead of the cells of the current Vframe. For example, we want to represent the structure g(h(W),Y,f(X,W)) and the bindings environment is the same Vframe-1. If hpt and fpt are pointers to the templates h(1) and f(1,2) respectively, the lowest level structures h(W) and f(X,W) will be represented as

| hpt | 1 | and | fpt | 3 | 1 |

where 1 and 3 are indices to the SVframe of higher level structure g(..). If sh and sf are pointer to the structures h(W) and f(X,W) respectively, the Stemp g(sh,2,sf), pointed to by gpt, will be the template for the Sframe-g2. Hence, the Sframe-g2 will be represented as

Sframe-g2 : | gpt | 1 | 3 | 2 |

where the numbers are indices to the Vframe-1.

Representation of lists follows similar principles. But it is important to appreciate the efficiency of the scheme for representing and sharing in the context of 'unification' and 'return' operations. In figure 13, we show two examples of unification that are general enough to illustrate the principle.

In figure 13(a), it is assumed that the variable X in P(X) is bound to a list [(a.Y.nil)|(b.Z.nil)], where Y and Z are unbound in QF. In this figure we show the relationships between the frames before and after unification. The Local frame (LF) for the clause (assuming two variables in the clause) is appended with two new cells to represent the imported unbound variables (corresponding to Y and Z). The negative indices in the Return frame (RF) are (as explained before) offsets in the LF. The representation scheme shows that the sublists (a.Y.nil) and (b.Z.nil) need not be copied though they contain unbound variables. The important point to note is that for lists of arbitrary length, number of additional cells created after unification depends on the number of unbound variables in the list before unification and is independent of the length of the list.

In figure 13(b), we illustrate another case of unification. The number of additional cells created for representing the bindings for the local frame is proportional to number of unbound variables in the concerned list. The list b.Z.c.nil is shared between the local frame and query frame. If the output logical variable Hq gets bound later during the processing of the clause, the only cell that gets affected is the fourth cell in LF and the list (St2) remains unaffected.

In figure 14, we show the list, bound to the output variable R, after the Return operator completes execution. Though the list (Sfa.Sfb.nil) contains unbound variables, representation of the list (shown boxed within dotted lines) is shared and appropriate modifications are reflected through the corresponding Sframe.

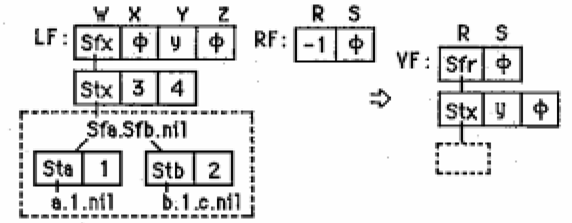


**Fig.14. Frame Relationships Before and After Unification** (sharing of sub-lists is indicated by the portions boxed)

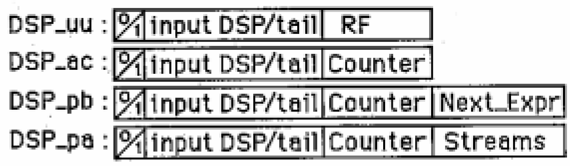


**Fig.15. Descriptor Format**

## 5. THE ABSTRACT ARCHITECTURE (LogDf)

The dynamic dataflow execution model [7,18] allows multiple tokens to be present on an arc of a graph and thus allows multiple parallel activations of the same dataflow graph. This allows multiple activations of a procedure, recursion or iteration. For example, in the PROCESS_EXP graph, multiple activations of the same SOLVE_EXP($E_i$) graph is required for different BE frames and these activations are overlapped in time. Each token flowing on the arc of a dataflow graph typically contains an identification of the destination node. In dynamic dataflow execution, the tokens related to different activations are identified by an additional piece of information attached to them, known as tag or color [7]. So a node in the dataflow graph fires (or becomes executable) when all the inputs receive tokens with the same color or tag.

In a typical single ring structure that implements the dynamic dataflow execution scheme [7,18] the tokens of the same color and same destination are matched in the matching unit to determine the firing condition of the node specified at the destination. The color of a token is changed[3] by an **Entry** unit associated with the input of the reentrant graph. The **Entry** unit stores the color of the input token and provides a new color. The corresponding **Exit** unit at the exit point of the graph restores the color stored by the **Entry** unit.

One of the major bottlenecks in the above mentioned structure is the matching unit. We will show in the following paragraphs how we eliminated the matching unit in the proposed abstract dynamic dataflow architecture for the parallel execution of logic programs.

From the graphs (shown in figures 2, 9, 10(a)-(d)), the following Entry-Exit pairs could easily be identified. The pairs are— **Unify** and **Return**, **PICK_Bcell** and **Scons** (level 1), **Activate** and **Scons** (level 2), **PAR_Activate** and **Cartesian Product**. Moreover, it should be noted that none of the **Entry**-type operators requires more than one input to be activated (another input, if any, is a constant). All the **Exit**-type operators require operands of matching color. In other words, all the **Exit**-type operators need to match operands corresponding to the same activation of the graph. The important characteristic of these graphs which may be noted is that one of the two operands for the **Exit**-type operators is produced by the corresponding **Entry**-type operator. For example, one of the operands for **Return** operator is the Return frame (RF) produced by the corresponding **Unify** operator. Moreover, only a single instance of this operand (RF) matches with a set of operands (of same color) provided as an S-stream of BE's at the other input. Similar property holds for all the other **Exit**-type of operators. For the **Scons** operator or the **Cartesian Product** operator the operand produced by the corresponding **Entry**-type operator is an S-stream descriptor.

The properties of the dataflow graphs mentioned above allow us to eliminate the matching unit. To eliminate the matching unit operation, the **Entry**-type operator allocates a descriptor in a descriptor store to hold one of the operands for the **Exit**-type operator. The address of the descriptor serves as the color for a particular invocation of a graph.

Two other characteristics related to the **Exit**-type operators are—

(1). Destination of the result packet is a prespecified operator type or may be determined from the color of the input token.

(2). None of the operators require a constant input.

The above characteristics of the **Exit**-type operator indicate that there is no need for an entry for the corresponding node in the node store. This property allows the result tokens out of the **Exit**-type operators to bypass the node store and reduce token traffic on the ring. The color of the input token is deposited in a specific field ('input DSP/tail' in figure 15) of the descriptor. The descriptor types created/used by each pair of the operators are as follows—

(1). DSPuu (**Unify—Return**)

(2). DSPac (**Activate—Scons**)

(3). DSPpb (**PICK_Bcell—Scons**)

(4). DSPpa (**PAR_Activate—Cartesian Product**)

The formats and the fields of these descriptors are shown in the figure 15. When an **Exit**-type operator receives an input token, the operator uses the color information in the token (i.e., the address of the descriptor) to retrieve the second operand from descriptor memory. Similarly the color of the output token and the destination of the result is obtained from the descriptor.

The basic structure of one of the rings of the abstract

---

[3] The description is intentionally kept at a nonspecific and abstract level.

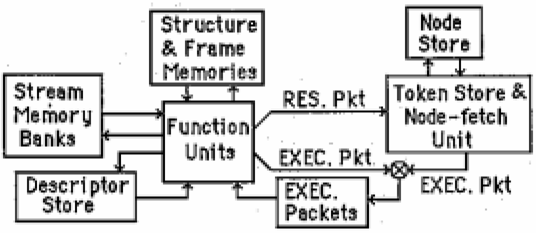


**Fig.16. One Ring of the LogDf Abstract Architecture** (Details of Arbitration/Distribution are not shown)

dataflow architecture (LogDf) for the proposed execution model is shown in figure 16. As the structure shows, the stream memory is spread over multiple banks and the controller is similar to an I-structure Memory controller[7] for dealing with non-strict operations. The distribution mechanism of S-streams over the banks and tokens over multiple rings play a significant role in the performance of this architecture. The mechanisms developed will be reported in a future paper.

The structure shown in figure 16 is self-explanatory. Node store representation of 8-queens program is shown in figure 17. Basically there are two types of instructions in the node store, namely, Activate and Pick_Bcell. We will explain the representation for one of these instructions. The operation code is shown in column #2. In the case of Activate, the entry in column #3 represents the goal template associated with the Activate node (cf. figure 1). Entry in column #4 is a pointer to a linked list of candidate clauses. The count of candidate clauses is indicated in column #5.

We will only explain how result packets as well as execution packets are produced by the function units. The function unit actually produces result packets (<color,destination,value>), but when the destination is an **Exit**-type operator, the destination is not in the node store. In such cases, the packet produced by the function unit is directly forwarded as an execution packet <color,opcode,value> to the execution packet queue.

The descriptor based execution mechanism is explained using one of the relatively complex functions. Let us consider the PROCESS_EXP graph with **PICK_Bcell** as the **Entry**-type operator and **Scons** as the **Exit**-type operator. When the **PICK_Bcell** receives a stream pointer as input, it creates a descriptor DSPpb as shown in figure 15. The color of the input token (eg. an address of DSPuu) is deposited in the 'input DSP' field of DSPpb and the address of this DSPpb becomes the new color of the result tokens produced by the operator.

Following the principles of operation described earlier, the contents of the various fields of the first input token (execution packet) received by a **Scons** function unit could be as follows— color: ↑DSPpb; value: pointer to a 2-level stream produced by one of the SOLVE_EXP graph; opcode: **Scons**. The **Scons** function will use the contents

<u>Source Code (8-queen) :</u>

```
C1 : queen([],Y,[]).
C2 : queen([X| L],Y,[C| Z]) :- select(C,[X| L],RC),safe(C,Y,1),
                               queen(RC,[C| Y],Z).
C3 : select(X,[X| Y],Y).
C4 : select(C,[X| Y],[X| RC]) :- select(C,Y,RC).
C5 : safe(U,[],W).
C6 : safe(U,[P| R],N) :- nodiag(U,P,N), M is N+1, safe(U,R,M).
C7 : nodiag(U,P,N) :- T1 is P+N, T2 is P-N, U\==T1, U\==T2.

     :- queen([1,2,3,4,5,6,7,8],[],Q).
```

<u>Node Store Representation :</u>

| | #1 | #2 | #3 | #4 | #5 |
|---|---|---|---|---|---|
| 1. | Slv_Qun_0 | : Activate | (ST0,[],V1) | Cl_Qun | 2 |
| 2. | Exp_Sel_1 | : Pick_Bcell | Slv_Sel_1 | Exp_Saf_1 | |
| 3. | Slv_Sel_1 | : Activate | (V4,ST1,V6) | Cl_Sel | 2 |
| 4. | Exp_Saf_1 | : Pick_Bcell | Slv_Saf_1 | Exp_Qun_1 | |
| 5. | Slv_Saf_1 | : Activate | (V4,V3,1) | Cl_Saf | 2 |
| 6. | Exp_Qun_1 | : Pick_Bcell | Slv_Qun_1 | Rtn | |
| 7. | Slv_Qun_1 | : Activate | (V6,ST3,V5) | Cl_Qun | 2 |
| 8. | Exp_Sel_2 | : Pick_Bcell | Slv_Sel_2 | Rtn | |
| 9. | Slv_Sel_2 | : Activate | (V1,V3,V4) | Cl_Sel | 2 |
| 10. | Exp_Ndg | : Pick_Bcell | Slv_Ndg | Exp_Add_1 | |
| 11. | Slv_Ndg | : Activate | (V1,V2,V4) | Cl_Ndg | 1 |
| 12. | Exp_Add_1 | : Pick_Bcell | Slv_Add_1 | Exp_Saf_2 | |
| 13. | Slv_Add_1 | : Add | (1,V4,V5) | | |
| 14. | Exp_Saf_2 | : Pick_Bcell | Slv_Saf_2 | Rtn | |
| 15. | Slv_Saf_2 | : Activate | (V1,V3,V5) | Cl_Saf | 2 |
| 16. | Exp_Add_2 | : Pick_Bcell | Slv_Add_2 | Exp_Sub | |
| 17. | Slv_Add_2 | : Add | (V2,V3,V4) | | |
| 18. | Exp_Sub | : Pick_Bcell | Slv_Sub | Exp_Neq_1 | |
| 19. | Slv_Sub | : Sub | (V2,V3,V5) | | |
| 20. | Exp_Neq_1 | : Pick_Bcell | Slv_Neq_1 | Exp_Neq_2 | |
| 21. | Slv_Neq_1 | : Noeq | (V1,V4) | | |
| 22. | Exp_Neq_2 | : Pick_Bcell | Slv_Neq_2 | Rtn | |
| 23. | Slv_Neq_2 | : Noeq | (V1,V5) | | |

| | | | | |
|---|---|---|---|---|
| 24. | Cl_Qun | : Cl | C1 | C2 |
| 25. | Cl_Sel | : Cl | C3 | C4 |
| 26. | Cl_Saf | : Cl | C5 | C6 |
| 27. | Cl_Ndg | : Cl | C7 | |

| | HEAD | H_Arg_No | Cl_Arg_No | BODY |
|---|---|---|---|---|
| 28. C1 | : ([],V1,[]) | 3 | 1 | null |
| 29. C2 | : (ST1,V3,ST2) | 3 | 6 | Exp_Sel_1 |
| 30. C3 | : (V1,ST4,V2) | 3 | 2 | null |
| 31. C4 | : (V1,ST5,ST6) | 3 | 4 | Exp_Sel_2 |
| 32. C5 | : (V1,[],V2) | 3 | 2 | null |
| 33. C6 | : (V1,ST7,V4) | 3 | 5 | Exp_Ndg |
| 34. C7 | : (V1,V2,V3) | 3 | 5 | Exp_Add_2 |

<u>Stemp Mem :</u>

| | | |
|---|---|---|
| 1. ST0 : [1,2,3,4,5,6,7,8] | | 2. ST1 : [V1\| V2] |
| 3. ST2 : [V4\| V5] | 4. ST3 : [V4\| V3] | 5. ST4 : [V1\| V2] |
| 6. ST5 : [V2\| V3] | 7. ST6 : [V2\| V4] | 8. ST7 : [V2\| V3] |

**Fig.17. Node Store Representation (8-queen)**

of the 'input DSP' field of the DSPpb as the second operand. In any case, the **Scons** function creates a new cell to store the first operand (the "value" in the input token) in the car field of the cell. Then retrieves the content of the second operand (↑DSPuu, in this case) and uses

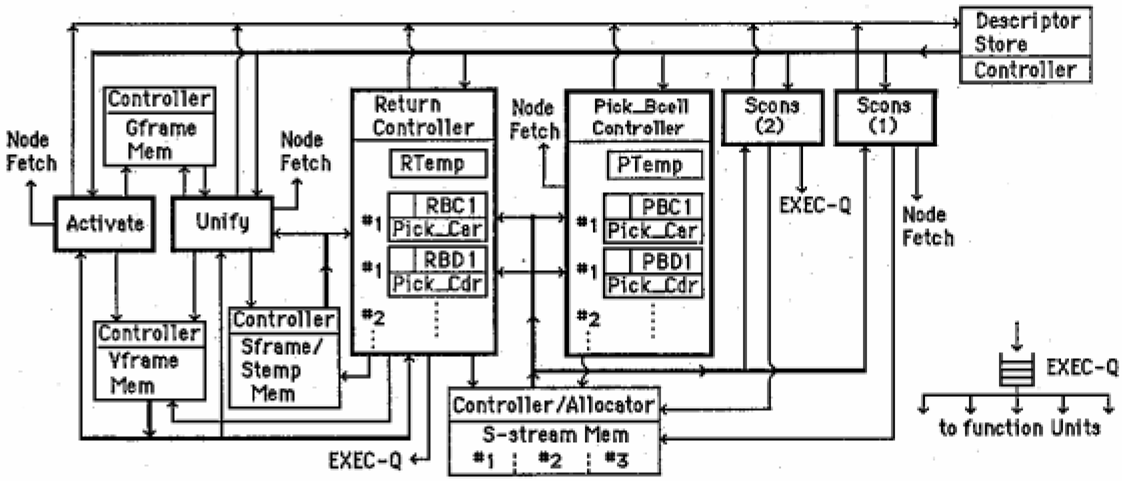


**Fig.18. Function Unit Organization for a Ring of LogDf**

it as the color of the output token. The result packet (a pointer to the new S-stream) is forwarded to the address obtained from the 'Next_Expr' field of the DSPpb.

Now we will consider the case when the input token received by the **Scons** function is not the first one of this color (identification is made by examining the flag in the first field of DSPpb.) In this case, a new cell will be created as before and appended to the 'tail' available from the DSPpb and the address of the new cell will be used to update the 'tail' field. No result token will be generated. The descriptor based technique helps in reducing token traffic on the ring significantly.

## 6. ORGANIZATION OF THE FUNCTION UNITS

The organization of the function units and various memories for one ring of the LogDf is shown in figure 18. We show one function unit of each type and their logical relationships with the memory components. The simulator uses varying number of these units. Due to brevity of presentation, the details of the operation of all the function units could not be included in this paper. The units operate in asynchronous fashion. The bold lines in the diagram indicate buses similar to Common Data Buses (CDB) in IBM 360/91. Read requests to any of these memories are tagged with an identifier for the destination and the tag is returned with the result.

A logic program is compiled into set of macro-l evel operations (as shown in the node store representation in figure 17) corresponding to the operators described in the previous sections. Additional parallelism is provided at the level of microoperations executed in each of the autonomous function units. The autonomous nature of the function units and elimination of the Exit-type operators from the node store reduces the possibility of queuing at the node store.

Three of the most important functions related to the S-stream based execution are the Pick Bcell, SCons and the Return functions. We will provide a brief description of these functions in the following paragraphs.

(i) **PICK_Bcell** : The input of the function is a pointer to an S-stream. The function could be described as a recursive composition of multiple 'Pick' functions. Pick functions are assigned to traverse the S-stream recursively to search for Bcells. The traversal is performed using the car and cdr pointers simultaneously. The operator is nonstrict and eager, as it starts performing the traversal even if only the car field of the cell is defined and the cdr field is undefined. The output of the PICK_Bcell operator is a sequence of pointers, each pointing to a Vframe (the pointers are obtained from the car fields of B-cells). A counter is associated with this operator. The counter is incremented every time the function outputs a pointer to a Vframe. The counter corresponds to the 'count' field of the associated DSPpb.

The function unit corresponding to this function contains a number of Pick_Car and Pick_Cdr modules that operate in asynchronous fashion. These modules perform the recursive decomposition of the S-stream to provide the B-cell outputs.

The register PTemp has a count field which keeps track of the number of independent stream traversal operations on the S-stream in progress. The count field is incremented by the Pick_Car module when it finds an S-cell. The counter is decremented on completion of the traversal of a branch (i.e., Pick-Cdr hits 'eos'). When the count field in PTemp becomes zero, it indicates the completion of the PICK_Bcell operation. The completion of the function is indicated by decrementing the count field in the descriptor by one. The registers PBC and PBD shown in the figure receive the data forwarded by the S-memory controller using the common data bus.

(ii) **Stream Cons (Scons)** : There is a descriptor (DSPac/DSPpb) associated with each **Scons**. The input[4] of **Scons** is a sequence of pointers, to streams or Vframes, or null values (fail). The function performed by the operator depends on whether the input is null or not. If the input is non-null, ie. a pointer, a new cell is allocated. The pointer is written into the car part of the cell and the address of the cell is written into the cdr part of the current tail (if any) as well as the 'tail' field of the corresponding descriptor. The count field of the descriptor is also decremented. If the input is null, the only operation performed by the operator is to decrement the count field.

The only output of this operator is the pointer to the first generated cell when the first non-null input is received. Another important function performed by this operator is the generation of end-of-stream ('eos') indicator for the stream under construction. The 'eos' indicator is written in the cdr part of a cell. The count field in the descriptor is used to determine 'eos' condition.

(iii) **Return** : The two inputs to this function are a pointer to the R-frame (created by **Unify**) and a pointer to an S-stream produced by the last body literal. The principle of operation is similar to the **PICK_Bcell** operator in the sense that it decomposes an input S-stream to select the Bcells. The additional functions performed by this operator are to create new Vframes for each Vframe selected and append the pointers (to the new frames) to create a single level stream. A new Vframe has the same content as the R-frame, except the cells with negative indices are updated to the bindings of the indexed cells in the received Vframe. As in the **Scons** function, the pointer to the stream is output once the first cell of the output stream is produced.

## 7. CONCLUSION

In this paper we have systematically developed the principle underlying the S-stream based data-driven model for parallel execution of logic programs. The S-stream structure introduced in this paper provides parallelism in construction/consumption of multiple binding environments in a non-strict eager fashion. The eager evaluation scheme is conservative compared to Amamiya's proposal [4] in the sense that a cell in the stream is not constructed prior to receiving a binding for the car field of the cell. This is particularly useful in conserving processing resources and improve processor utilization. The relationships between the descriptors, the levels of the S-stream and the processes provide a logical framework for controlling explosive parallelism which might result in an unrestricted AND/OR parallel execution of logic programs.

Optimal grain of parallelism is also a major concern in this project. Preliminary studies of a fine-grained model (similar to the one proposed by Ito et.al. [21] ) indicated that it was desirable to use macro operators at the dataflow graph level and provide parallelism within these operators using autonomous function units capable of exploiting microoperation level parallelism.

A simulator and a compiler for an earlier version (OR-parallel) was designed for an architecture similar to the Manchester dataflow machine. Performance evaluation was done for a number of programs [10]. The operand matching unit and the distribution scheme were found to be the bottlenecks. The lessons learnt from that experience led to this current LogDf model. The simulator for a single ring of LogDf is just ready at the time of this reporting. It proves the correctness of the execution scheme. The performance results could not be included at this time. We hope to report the results in the near future.

The ultimate goal of this project is to design a multi-ring architecture to solve a number of top-level queries in parallel. We are studying the effect of different distribution strategies for S-streams and descriptors on a multi-ring structure. Developments of techniques for controlling parallelism and dynamic load balancing are also in progress.

---
[4] To simplify the figures, the symbol of Scons shown represents multiple instances of the Scons operation using a single stream descriptor.

## ACKNOWLEDGEMENTS

We would like to thank Doug DeGroot for his valuable comments on an earlier draft of this paper.
Thanks are also due to Cheri Dowell for her help in designing the earlier version of the compiler and the simulator.

## REFERENCES

[1] Amamiya, M., and R. Hasegawa, "A List-Processing-Oriented Data Flow Machine Architecture," Proc. of the 1982 National Computer Conference, AFIPS, pp. 143-151, 1982.

[2] Amamiya, M., R. Hasegawa, and H. Mikami, "List Processing with Data Flow Machine," Lecture notes in Computer Science, No. 147, Springer-Verlag, pp. 165-190, 1983.

[3] Amamiya, M., and R. Hasegawa, "Dataflow Computing and Eager and Lazy Evaluations," New Generation Computing, Vol. 2, No. 2, pp. 105-129, 1984.

[4] Amamiya, M., and R. Hasegawa, "Parallel Execution of Logic Programs Based on Dataflow Concept," Proc. of the International Conference on Fifth Generation Computer Systems, ICOT, pp. 507-516, 1984.

[5] Amamiya, M., M. Takesue, R. Hasegawa, and H. Mikami, "Implementation and Evaluation of a List-Processing-Oriented Data Flow Machine," 13th Int'l. Symp. on Computer Arch., pp. 10-19, Tokyo, June 1986.

[6] Arvind, and R. A. Iannucci, "A Critique of Multiprocessing von Neumann Style," ACM Proc. of the 10th

Int'l Symp. on Comp. Arch., pp. 426-436, June 1983.

[7] Arvind, "The Tagged Token Dataflow Architecture," CSG Memo-229, MIT lab. for Comp. Sc., July, 1983.

[8] Barahona, P. M. C. C., "Sepcification and Control of Execution of Nondeterministic Dataflow Programs," Ph.D. Thesis, Dept. of Comp. Sci., Univ of Manchester, May 1987,

[9] Bic, L., "Execution of Logic Programs on a Dataflow Architecture," 11th Annual Symp. on Computer Architecture, pp. 290-296, 1984.

[10] Biswas, P. and C. C. Tseng, "Preliminary Evaluation of a Stream based Data-driven Model for Parallel Execution of Logic Programs," TR 88-21, Tech. Rep., Dept. of Comp. Sci., SMU, Dallas, TX, 1988.

[11] Biswas P. and S. C. Su, "A Scalable Abstract Machine Model to Support Limited-OR/Restricted-AND Parallelism in Logic Programs," Proc. of 5th Int'l Conf. Symp. on Logic Programming, pp. 1160-1179, August 1988.

[12] Chang, J. H., A. M. Despain, and D. DeGroot, "And-Parallelism of Logic Programs Based on a Static Data Dependency Analysis," Proc. COMPCON'85, pp 218-226, 1985.

[13] Conery, J. S., "The AND/OR Process Model for Parallel Interpretation of Logic Programs," Tech. Rep. 204, Dept. of ICS, Univ. of Calif., Irvine, 1983.

[14] Conery, J. S., "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors," Symp. on Logic Programmimg, pp 457-467, Sept. 1987.

[15] Crammond, J., "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages," IEEE Transactions on Computer Vol. C-34, NO 10, Oct. 1985.

[16] DeGroot, D., "Restricted AND-Parallelism," Proc. of the Int'l. Conf. on 5th Generation Computer Systems, pp 471-478, ICOT, 1984.

[17] Friedman, D. P., and D. S. Wise, "CONS Should Not Evaluate its Arguments," Automata, Languages and Programming, Edinburgh Univ. Press, pp. 257-284, 1976.

[18] Gurd, J.R., C. C. Kirkham and I. Watson, "The Manchester Prototype Dataflow Computer," CACM, Vol 28, No. 1, Jan., 1985.

[19] Halim, Z., "A Data-Driven Machine for OR-Parallel Evaluation of Logic Programs," New Generation Computing, Ohmsha, LTD and Springer-Verlag, Vol. 4, pp. 5-33, 1986.

[20] Hogger, C. J., "Introduction to Logic Programming," Academic Press, 1984.

[21] Ito, N., and H. Shimizu, "Dataflow Based Execution Mechanisms of Parallel and Concurrent Prolog," New Generation Computing, No. 3, pp. 15-41, 1984.

[22] Li, P. Y. P., and A. J. Martin, "The Sync Model for Parallel Execution of Logic Programming," Third IEEE Symp. on Logic Programming, Sept. 1986.

[23] Sargeant, J., "Load Balancing Locality and Parallelism Control in Fine-Grain Parallel Machines," Tech. Rep. #UMCS-86-11-5, Univ. of Manchester, Jan. 1987

[24] Tseng, C. C. and P. Biswas, "A Data-driven Parallel Execution Model for Logic Programs," Proc. of 5th Int'l Conf. Symp. on Logic Programming, pp. 1204-1222, August, 1988.

[25] Umeyama, S., and K. Tamura, "A Parallel Execution Model of Logic Programs," 10th Int'l. Symp. on Computer Architecture, pp. 349-355, 1983.

[26] Warren, D. H. D., "OR-parallel execution models of Prolog," In TAPSOFT'87, The 1987 Int'l Joint Conf. on Theory and Practice of Software Development, Pisa, Italy, pp. 243-259, Springer-Verlag, March 1987.

[27] Wise, M. J., "Parallel Prolog: the Construction of a data driven model," ACM Symp. on Lisp and Functional Programming, pp. 56-66, 1982.

[28] Wise, M. J., "Prolog Multiprocessors," Prentice-Hall, 1987.