ICOT Technical Report: TR-033

TR-033

Parallel Inference Machine Based on the Data Flow Model

by
Noriyoshi Ito and Kanae Masuda

December, 1983

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

Parallel Inference Machine Based on the Data Flow Model

Noriyoshi Ito, Kanae Masuda

Institute for New Generation Computer Technology

1. INTRODUCTION

Equipped with basic functions such as pattern matching and non-deterministic control, a logic programming language like Prolog (Programming in Logic) [13] seems suitable for use in knowledge information processing systems. For such inference-based processing systems, generally, a sequence of processing is not predetermined in advance; such systems, therefore, require a function to carry on processing, while heuristically seeking appropriate procedures. This function is inherent in Prolog.

Prolog provides the potential for parallel processing [7]. Conventional sequential processing systems are based on the depth-first search approach, or the sequentially controlled non-deterministic search approach with backtracking. By contrast, the breadth-first search approach, simultaneously trying multiple possible searches, can solve problems faster by initiating a number of searching processes on multiple processors.

The data flow model can naturally implement parallel processing [1] [2] [8], and its execution mechanism is closely related to that of Prolog. This paper describes an execution mechanism and a brief machine architecture based on the data flow model to implement parallel execution of Prolog. First, an execution mechanism of Prolog is discussed in Section 2, followed by a description of the parallel unification model in our machine in Section 3. Section 4 explains primitive operators for non-deterministic merging and unification. Finally, the architecture of a parallel Prolog machine is outlined in Section 5.

2. EXECUTION OF PROLOG

Unification, or pattern matching, is a basic function of Prolog. A Prolog program is executed by repeatedly performing unifications between goal statements which correspond to queries and a set of clauses which correspond to given knowledge.

Each clause consists of a head part and a body part as follows:

P <- Q1 & Q2 & ... & Qn. (n>=0)

where P denotes a head literal, Qi denotes a body literal, and the symbol '<-' means implication.

When clauses are restricted to Horn clauses as in Prolog, the head consists of at most one literal. A clause without a head is called a goal statement. The body is also optional; a clause without a body is called a unit clause; otherwise, a clause is called a non-unit clause. If there are multiple literals in the body, they are connected through ANDs.

When a goal statements has multiple literals, these literals which are called goal literals are ANDed. The unification of the whole goal statement, therefore, can not succeed unless it succeeds for all the literals in that goal statement. That is, the goal statement cannot be successfully solved unless all the literals in the goal statement can be solved.

A clause which is potentially unifiable with a goal literal must have a head literal which has the same predicate as the goal literal. A subset of such clauses is referred to as the definition of that predicate. A set of clauses constituting a definition are ORed. In other words, a goal literal can be solved if successfully unified with at least one clause.

When a goal literal in a goal statement is given, the primitive unification operation is executed by invoking one of the clauses in the definition of the goal literal's predicate, and then unifying (i.e. pattern-matching) between the goal literal and the head literal of the selected clause. This operation will

produce a common instance of two literals, if the pattern-matching has succeeded; or a 'fail' signal, otherwise.

When a given goal literal is successfully unified with the head literal of a non-unit clause, then another unification process is initiated taking its body as a new goal statement. When the goal literal is successfully unified with the head literal of a unit clause the unification operation terminates (i.e. the solution is obtained); then the result is returned to the parent process which called the unification process.

3. PARALLEL UNIFICATION MODEL

In Prolog, the parallelism involved in the unification can be divided into three types:

- OR parallelism
- AND parallelism
- parallelism among arguments

These are described below.

3.1 OR Parallelism

Given a goal literal and the definition of its predicate, unification between the goal literal and the definition can be performed in parallel on all clauses in the definition.

A sequential Prolog interpreter system uses backtracking to control unification, and the order for calling clauses in a definition is predetermined. When a goal literal is given, the first clause is invoked according to the order and unified with the goal literal. When the unification fails (i.e. no solution exists), the backtracking mechanism invokes the next clause in the order and unification is retried on this clause. On the other hand, our parallel Prolog machine aims at high-performance inference processing by performing unification in parallel on ORed clauses in the definition.

This OR parallel execution can be performed simply by simultaneously initiating unification for individual clauses in the definition of the goal predicate. Then, the successful unification results (i.e. solutions) are merged to form an output. When only a single solution is required, one of the successful solutions, perhaps first one obtained, can be output. This "don't care" non-determinism can be implemented by the guarded clause mechanism described in [10]. The merge operation generally outputs solutions in a non-deterministic order; it may output them in the order obtained. Our machine will employ structured data called a stream, which plays a role of a "pipe" through which merged solutions are delivered [3] [14].

The stream is a non-strict structured data and provides means of asynchronous communications between producer processes, which generate the elements of a stream, and consumer processes, which refer those elements. For our machine, producer processes correspond to unification processes operating on an OR-parallel basis and generating merged solutions, while consumer processes correspond to other unification processes which input the merged solutions. The stream will be empty when all the OR-parallel unifications of the goal literal have failed.

3.2 AND Parallelism

ANDed literals in a goal statement can be solved in parallel. This AND parallelism, however, involves the following problem: when goal literals being executed on an AND-parallel basis have shared variables, the solutions for these variables must be consistent. In checking for this consistency, another unification operation, such as a join operation in a data base system, must be performed on sets of the solutions for the literals with shared variables. When these sets of solutions are large, then the consistency checking operation tends to require longer overhead and larger amounts of resources. For this reason, we will introduce AND parallelism into our machine making use of the pipeline effect as described later.

3.3 Parallelism Among Arguments

When a goal literal and one of the head literals of its unifiable clauses consist of multiple arguments, the unifications between the arguments in the goal literal and the corresponding position's arguments in the clause head literal can be performed independently by representing the unification procedure as a data flow graph. When an argument of the goal literal and the corresponding position's argument of the head literal are structures, then the parallel unification of their substructures can also be implemented.

4. PRIMITIVE OPERATORS FOR UNIFICATION

Prolog programs are build from terms. A term is either a symbol, an integer, a variable, a list, or a vector. In addition to these data types, we introduce another data type, which is called a stream. Of these, structured data - a list, vector, or stream - is represented as a pointer to structure memory. To improve unification performance, our machine employs a tagged architecture, and represents data using a tag field showing its data type and a value field.

In this section, we will first describe the basic unification primitive operators between any two terms used for parallel unification among arguments. Next, we will explain the non-deterministic stream merging primitive operators which are used when implementing OR parallelism. And finally, we will describe the consistency checking operations among AND literals.

4.1 Basic Unification Primitives

The basic unification primitive, 'term-unify', is used in the unification between one argument in a goal literal and the corresponding position's argument in the head literal of its definition. This primitive can be represented by a data flow graph shown in Figure 4.1. As the figure shows, this primitive inputs a pair of arguments and returns a common instance of them. When one argument is a variable, it returns the other. When both arguments are atoms (symbols or

integers), it returns the unification result of two arguments. (When they are the same atom, it returns the atom itself; otherwise, it returns the special atom 'fail', which means the unification has failed.) When both are lists or vectors, it calls the list- or vector-unify primitive, respectively, which, in turn recursively calls the term-unify. In other cases, the term-unify returns 'fail'.

When introducing parallelism among the arguments described above, all the independent unification results must be checked for consistency, that is, the literal unification succeeds only if all the argument unifications have successfully completed. The required consistency checking enables the parallelism among arguments to be considered as a variant of AND parallelism.

If the goal literal has shared unbound variables as its arguments, however, the consistency checking must guarantee that the same shared variables are bound to the same instances. Some examples of this are given below. Suppose that a goal statement

```
<- ... & p(Z,Z) & ...
```

is given, and the instance of the variable Z is a non-ground term (i.e. it is an unbound variable or includes some unbound variables) before p(Z,Z) is called, and also suppose that a clause of definition p

```
p(f(U),g(V)) <- ...
```

is given. The unification process of this clause must have some way of knowing that its input arguments have some shared unbound variables in common with each other.

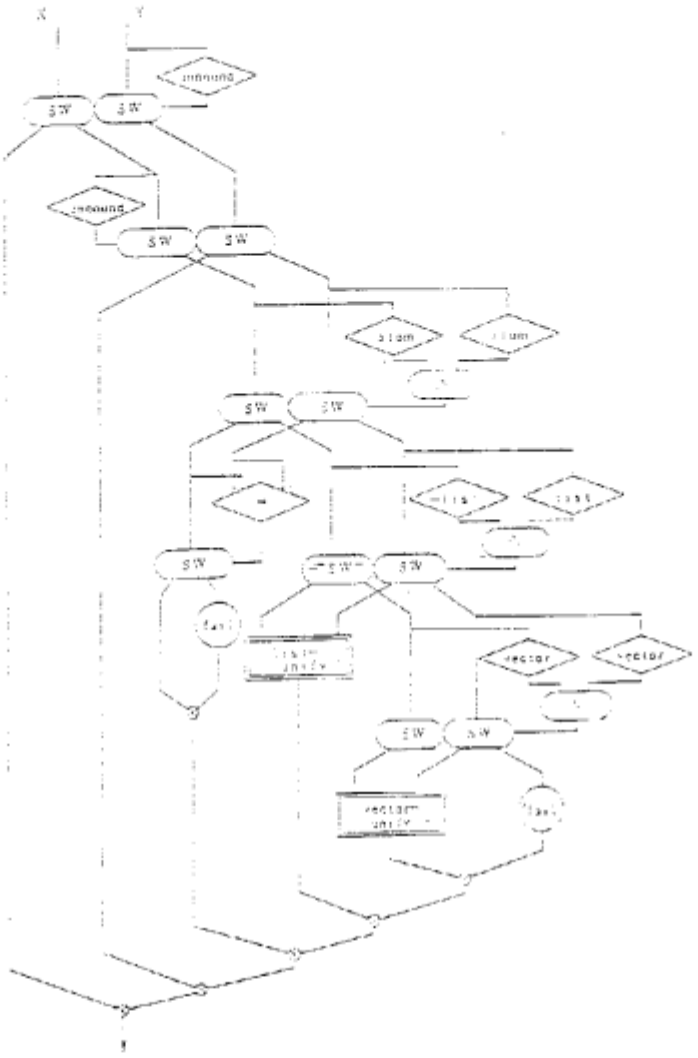


Figure 4.1 Data Flow Graph of a 'term-unify' primitive

We will use a share operator to change the unbound variables in the instance of Z to the shared unbound variables, prior to calling the definition of p. In order to perform this operation faster, the data type field of the instance has an unbound flag, which is set ON when the instance is an unbound variable or when it has any unbound variable as its substructure. So the structure construction operator, which is used to construct new structured data from existing data, will check all the substructure's unbound flags, and will produce a new data structure with an unbound flag which is either ON or OFF, depending on the status of the old flags. The share operator executes the following procedure:

(a)If the unbound flag of the share operator's argument is OFF, the operator returns the argument itself.

(b)If the flag is ON, and if the argument is an unbound variable, the operator returns a new shared unbound variable, which has a unique shared unbound variable name in its data part.

(c)If the flag is ON, and if the argument is a shared unbound variable, the operator returns the argument itself.

(d) If the flag is ON, and if the argument is a structure, the operator recursively calls the share operations to be performed on all its substructures, and returns a new structure constructed of their results.

The unify operation of the clause can be represented by the data flow graph in Figure 4.2. The unify operators in the graph are somewhat modified versions of the above 'term-unify' primitive. The graph shows that the unify operator of each argument has two outputs; one is an instance between an argument of the goal literal and its corresponding position's argument of the head literal, and the other is a binding environment of the shared unbound variables included in the goal argument. The binding environments, which consist of the lists of the unifiers of the shared unbound variables (i.e. the shared unbound variable names and their instances), are sent to a set of consistency check operators.

A consistency check operator receives a pair of the binding environments, and constructs a new binding environment as follows:

(a) If one of the input environments is a 'fail' atom (i.e. if its unification has failed), then the operator outputs 'fail' as a new environment.

(b) If one of the input environments is 'nil' (i.e. if its goal argument has no shared variable), then the operator outputs another input environment as a new environment.

(c) If both environments are lists, then the operator checks consistency between the two environments; it gets the first unifier from the first environment, and searches the second environment for the same shared unbound variable as the one in the first unifier; if the associative searching succeeds, the two instances in the first unifier and the unifier in the second environment are checked for consistency, i.e. these two instances are unified; this unification will produce 'fail' if the the unification has failed, or a new instance for the shared unbound variable if the unification has succeeded.

The above consistency check operation is executed for all the unifiers in the first environment, and will produce a new environment for all the shared variables included in both of the input environments. This environment includes the most general unifiers of the shared variables.

On the other hand, the unify operator for the arguments will execute the following procedure; it tests the goal argument's data type for shared unbound variable. The data type field has a shared flag, which indicates whether or not the data has any shared unbound variable as its substructure, just like the unbound flag described above. If the flag is OFF, the operator produces two outputs: one is a common instance between the goal literal's argument and the head literal's argument, and the other is a 'nil' value which shows a binding environment is empty. If the shared flag is ON, the operator produces the following two results:

(a) If the goal literal's argument is a shared unbound variable, it produces the shared unbound variable as the instance, and a new structure cell address as its unifier. This cell is used to store the shared variable and the contents of the head argument, which is an instance of the variable.

(b) If the goal literal's argument is a structured data, it decomposes the structure into substructures, executes unifications on these substructures by recursively calling the unify operators, and constructs two structures from these unification results: one is a construction of the substructure's instances, and the other is a new environment.

If a goal literal has shared unbound variables, the instance of the unification between the goal literal and head literal includes the shared unbound variables themselves. The instance of these variables may be gotten from the binding environment, by again associatively searching the environment with shared variable names as keys. This operation may be done by the subs (abbreviation of substitute) operator in the above graph.

4.2 Stream Merging Primitives

Another basic primitive is a merge operation of a non-deterministic stream. In order to implement a stream as an arbitrary incremental structure, a stream body is represented by a list type data structure; it has a first part where a stream element is stored, and a rest part where a stream rest pointer is stored. This merge operation can be implemented as follows. Before invoking clauses of the goal literal's definition, a create-stream operator is executed; this operator returns two outputs: one is a pointer of a stream head cell, and the other is a pointer of a stream tail pointer cell. The stream head cell is the beginning of the stream, and its pointer is sent to the consumer processes, which can get stream elements from the pointer. The stream tail pointer cell is

initialized to point to the stream head cell, and is shared by the invoked OR-parallel processes. Each OR process, if successfully terminated, appends a new solution to the end of the stream by updating the stream tail pointer to point to the new stream end. The solution of the OR process is represented by a list of the final instances for the input arguments of the head literal and a binding environment of the shared variables.

This append-stream operation executes the following cycles: it allocates a new stream body cell, reads the contents of the stream tail pointer, updates it to point to the new stream body cell, writes a solution to the first part of the body cell, and finally writes the pointer of the body cell to the rest part of an old stream tail cell. As the stream tail pointer cell is shared by the OR processes, the append-stream operator must lock the cell against other append-stream operators while updating the contents of the cell. An append-stream operator of a failed OR process performs no operation, simply decrementing the reference count of the stream tail pointer cell.

Our machine adopted the refernece count method for the memory management[6], which is also used for the stream manegement; when all the OR processes of the stream have terminated (i.e. when the reference count of the stream tail pointer cell has reached zero), the final operator writes 'fail', which is used for a symbol of end-of-stream, to the rest part of the stream tail.

4.3 Consistency Checking Operations Among AND Literals

When a given goal statement has multiple literals with shared variables, the execution order of these literals is determined according to certain rules. The order must be determined uniquely, for example, by selecting literals from left to right, or by specifying the input/output relation of variables like in the Relational Language by K.L. Clark and S. Gregory [4].

Literals without shared variables do not have to be solved on a pipeline basis; their unifications can be executed in parallel. This approach is based on the idea to improve parallelism, and also to avoid the duplication of unifications.

Suppose that a goal statement

<- p(X) & q(Y) & r(X,Y).

is given, where p, q, and r are predicates, and X and Y are variables. As the literals p(X) and q(Y) have no shared variable in common, they can be solved independently. If a sequential execution method of AND literals is used, and if the p(X) produces multiple instances of variable X, then the same unification of the literal q(Y) must be applied for all the results of the unification of literal p(X). In this approach, however, the unification of q(Y) may be applied only once, and is initiated without waiting for the results of the unification of p(X).

In the above goal, p(X) and q(Y) become the producers of the instances for variables X and Y, respectively, and r(X,Y) becomes a consumer of these streams. The unification process of r(X,Y) inputs two streams for variables X and Y, and performs unification on all the combinations of their instances. This can easily be achieved by the recursive calling of the literal p(X,Y) and by unfolding both streams to their elements.

This approach, however, involves some problems. Suppose that another goal statement below is given:

<- p(X,Y) & q(X) & r(Y)

In this case, when one of the solutions to the goal literal p(X,Y) is obtained, the instances for variables X and Y are sent to q(X) and r(Y), respectively. Then, q(X) and r(Y) will be solved independently.

However, if the instances of X and Y are non-ground terms (i.e. if they include some unbound variables before calling p(X,Y)), and if a unit clause below is given as the definition of p:

p(Z,Z) <-

where the head literal has multiple occurrences of the variable Z. Then the returned instances from p(Z,Z) may also be non-ground terms, and be bound to the same unbound variable Z.

This means that, when the execution of q(X) causes X to be bound to the some instances, the execution of r(Y) is affected by it (i.e. involves a side effect operation between q(X) and r(Y)).

It is difficult to detect such a side effect unification and to determine the execution order of literals at compile time. Therefore, operators that detect whether the instances from the literal p(X,Y) include the shared unbound variables or not, and those that check the consistency of these shared unbound variables, will be necessary.

The unification procedure of a clause where its head literal has multiple occurrences of the same variable, like p(Z,Z), is represented by the data flow graph in Figure 4.3.

Since an unbound variable can be unified with any form of term, the unify operators between two input arguments and the unbound variable Z can be omitted (these operators are shown by the broken lines). In order to guarantee the identity of the first and second arguments of predicate p in the above clause, the independent instances of their unifications must be unified again. This unify operator will produce an instance of variable Z, which is sent to the body of the clause, if the body exists. After the body unification has been completed, or no body exists, the share operator checks the final instance of Z as to whether or not it is a ground term.

In the given goal statement, therefore, the goal literal p(X,Y) produces the two instances for variable X and Y, which are the same unbound variable Z. These instances are sent to the new goal literals q(X) and r(Y), each of them will produce the instance Z and the binding environment of Z. Two binding environments of Z are checked for consistency by using consistency check operator described above. And, finally, two instances Z are substituted by the result of consistency check operator.

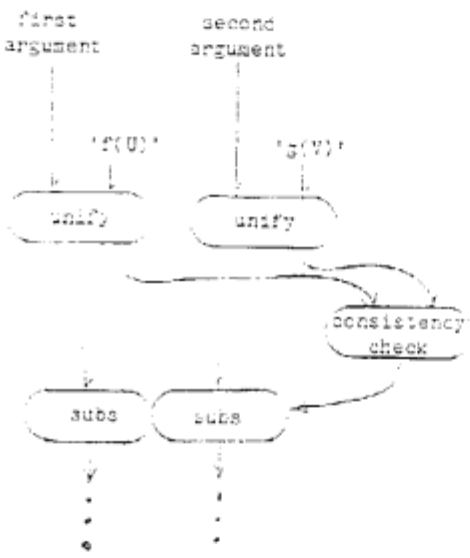


Figure 4.2 Data Flow Graph of the Clause p(f(U),g(V)) <- ...

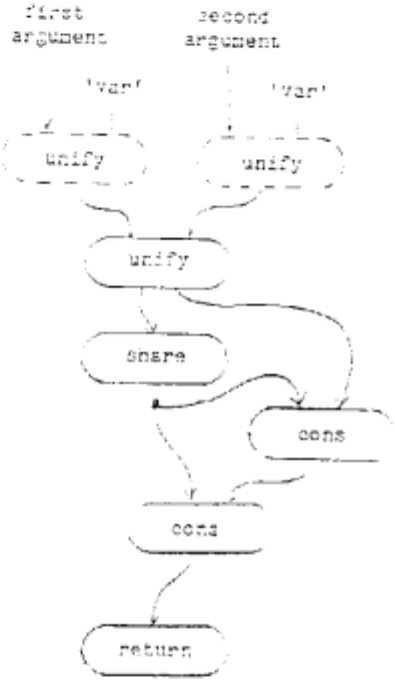


Figure 4.3 Data Flow Graph of the Clause p(Z,Z) <-

Figure 4.4 shows another example of the compiled code of a Prolog program represented by the data flow graph. This example program is a definition of the append shown below:

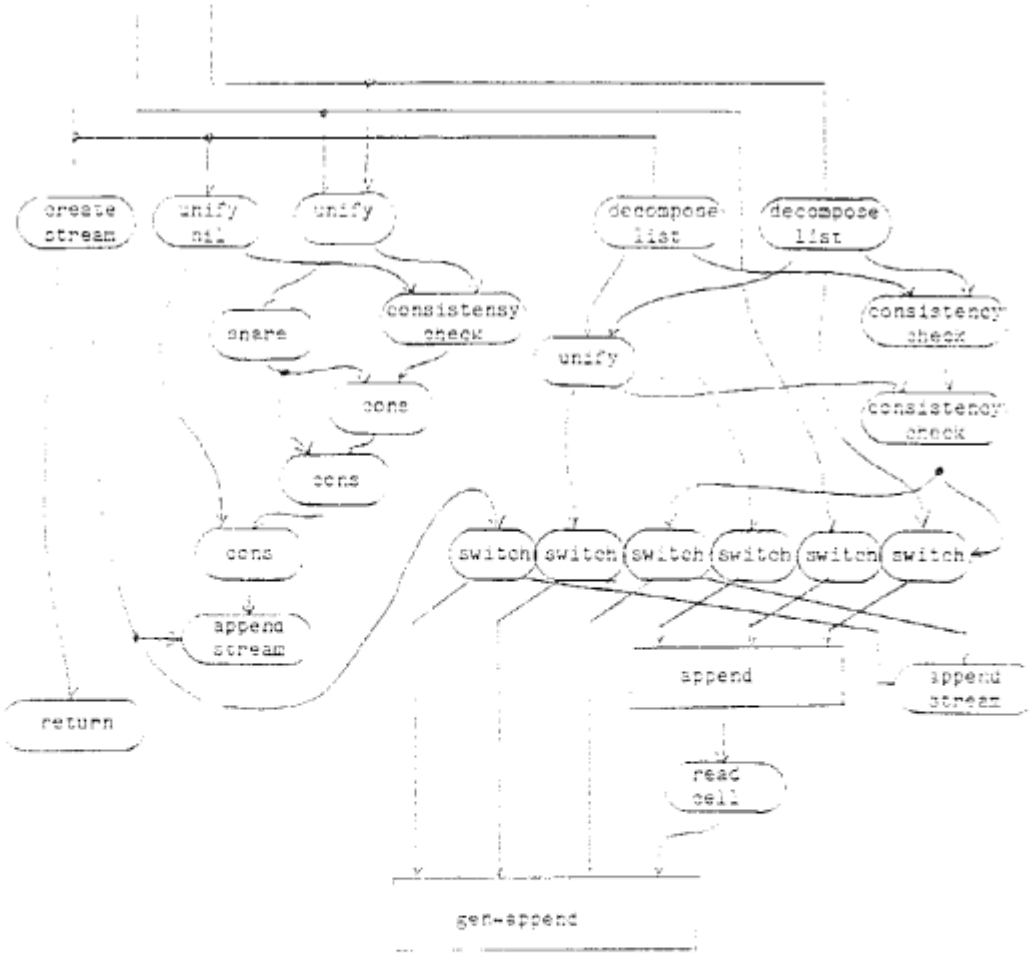
```
append([],X,X) <- .
append([H|X],Y,[H|Z]) <- append(X,Y,Z).
```

In this figure, a rectangular block means a procedure invocation, and the above-described 'term-unify' can be replaced with more efficient primitives, such as unify-with-nil, which unifies the input with 'nil', or the decompose-list operator, which unifies the input argument with the list and, if

this succeeds, decomposes it to left part and right part; if the input argument is an unbound variable, the left and right parts are also unbound variables.

Figure 4.4 (a) is a graph of append where the first and second clauses are executed in parallel, and Figure 4.4 (b) is a graph of gen-append, which is invoked in the second clause of the definition. The second clause calls append recursively, which generates a stream. The gen-append graph unfolds this stream into elements, and generates a new stream.

After the unifications of the head literal's arguments have been completed, their results are tested to see whether or not all the unifications have succeeded. In the first clause of the append definition, which is a unit clause, this operation is executed by using cons (construct) operators, the



Data Flow Graph of append

Figure 4.4 (a) Data Flow Graph Example

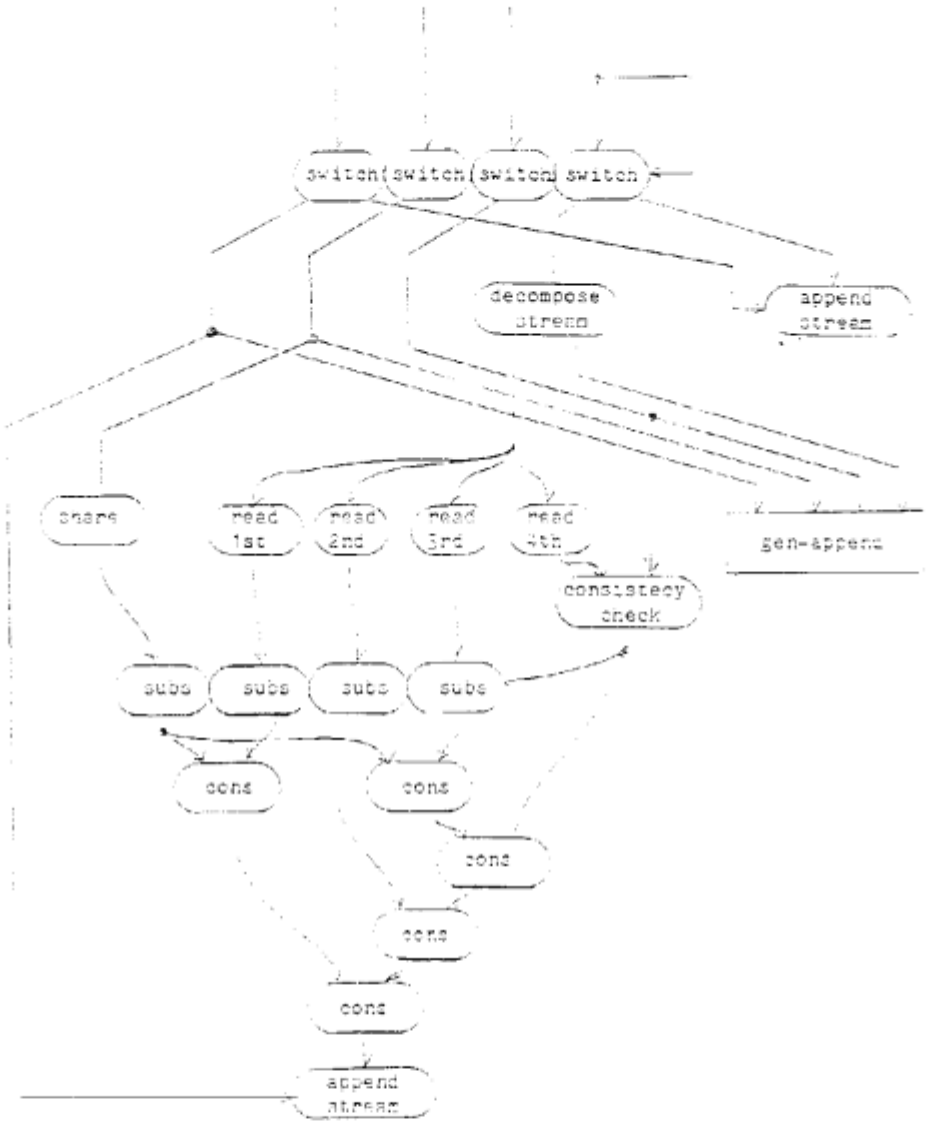


Figure 4.4 (b) Data Flow Graph Example

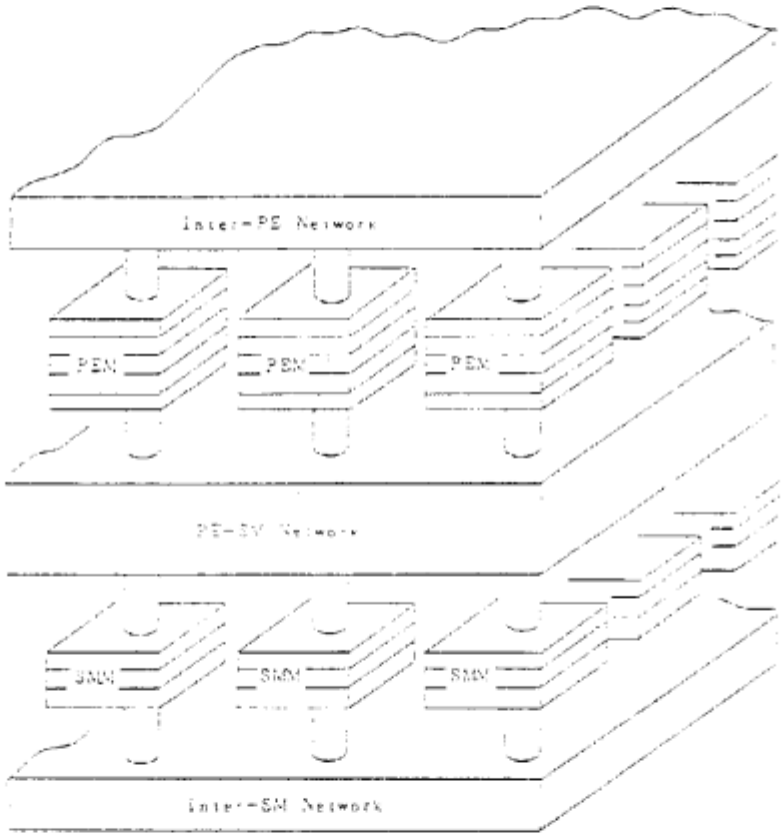
inputs of which are tested, and if all of them are not 'fail', generate a construction of them; otherwise, this operation generates a 'fail'. The output of cons operators initiates an append-stream operator, described above, which in turn appends the result to the end of stream.

In the second clause, which is a non-unit clause, the head literal's unification results are passed to its body if all the argument unifications have succeeded. A switch operator passes its top input to the left output port, if its right input is not a 'fail'; and passes its top input to the right output

port, otherwise.

5. MACHINE ARCHITECTURE AND ITS PERFORMANCE

The machine is constructed by multiple Processing Element Modules (PEMs) with multiple shared Structure Memory Modules (SMMs) as shown in Figure 5.1. These modules can operate independently of each other, and are connected by asynchronous communication networks; Inter-PE Network, PE-SM Network, and Inter-SM Network.



PEM: Processing Element Module
SMM: Structure Memory Module

Figure 5.1 Machine Architecture

Each PEM consists of an RPQ (Result Packet Queue), ICU (Instruction Control Unit) and EXU (Execution Unit). These units in the PEM constitute a circular pipeline structure and they send packets to the next-stage units via asynchronous communication interfaces. The RPQ is a first-in first-out memory which pools result packets(tokens) arrived, and sends them to the ICU. The ICU

receives result packets from the RPQ, detects whether the instructions are executable or not, and sends executable instructions to the EXU. The EXU is made up of multiple APUs (Atomic Processing Units), and distributes the received instructions to the APUs. The APUs execute the instructions and generate new result packets which in turn are sent to the next RPQ, or generate structure memory handling commands sent to the SMM via the PE-SM Network if structure memory operations are needed.

A unification procedure represented by a data flow graph is loaded in one of the PEMs, and executed. The PEM detects the executable instructions in the graph, and interprets them in parallel. If multiple procedure invocations are issued, these procedure instances may be distributed among the PEMs. Each PEM can execute multiple instances of the procedures when many procedures are activated. These instances are distinguished by the PEM number and their process identifiers.

The structured data is stored in the SMMs, which is accessible from the PEMs. The structured data, therefore, is represented by a pointer to the storage in the SMM, and can be shared by the multiple unification processes.

We have developed a software simulator for this machine, which is an event-driven, functional-block-level simulator, and used it to evaluate the performance of the machine. Each functional block corresponds to the units just described above. Some of their assumed operation and delay times are as follows: delay time in RPQ is 8 machine cycles, add instruction time in an APU is 3 machine cycles, result packet construction time and send time in an APU is 4 cycles, switching delay time of each switching node in the networks is 10 machine cycles, and so on.

Figure 5.2 shows the machine's simulation results when the number of modules(PEMs and SMMs) changes from one to four, where LIPS (Logical Inferences Per Second) are computed assuming a machine cycle speed of 250 nano second.

Two versions of a factorial program, which uses the divide-and-conquer method for increasing parallelism, are simulated: one is a non-stream version and other is a stream version. The other two programs are stream versions. We found that the non-stream version of the factorial program, where each predicate call is deterministic and is regarded as a pure function, shows a performance improvement by a factor of 2 to 3 over the stream version, which uses the stream control primitives described above.

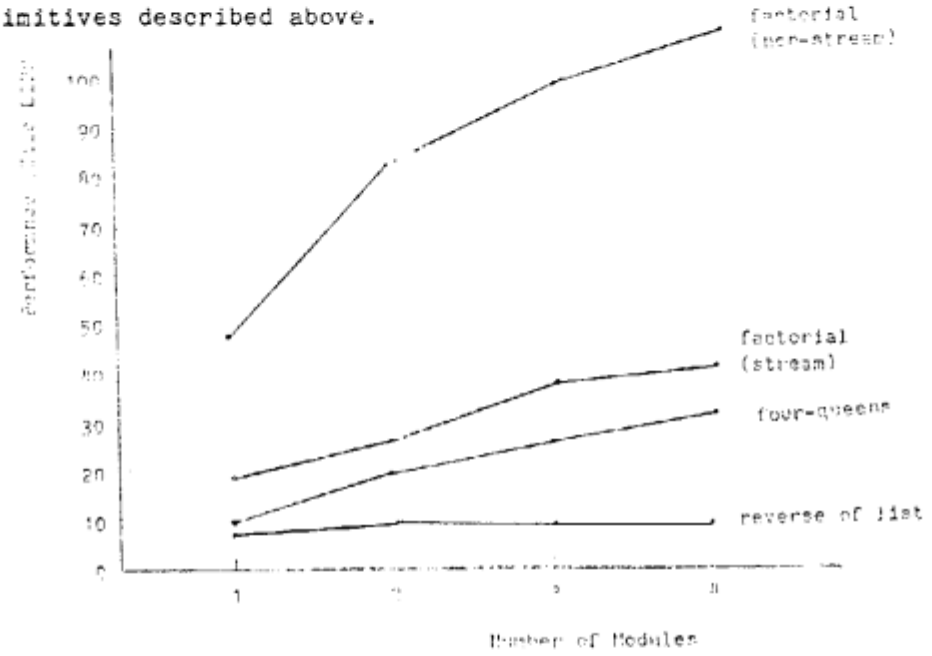


Figure 5.2 the Performance of the Machine

6. CONCLUSION

This paper has described a processing model of a data-flow-based Prolog machine and its architecture. In the description, we have shown that the introduction of a stream concept can implement OR-parallel and AND-parallel processing of Prolog. The machine simulation results show that our machine has the potential of being used as a high-performance inference machine. At present, we are simulating a larger scaled machine and designing a more detailed structure.

Acknowledgments

We would like to thank Dr. Kunio Murakami, Chief of the First Research Laboratory at ICOT, and other ICOT research members for their valuable comments.

References

[1] Amamiya,M. and R.Hasegawa, "Data Flow Machine and Functional Language", AL81-84, PRL81-63, the Institute of Electronics and Communication Engineers of Japan, Dec. 1981 (in Japanese).

[2] Arvind, K.P.Gostelow and W.E.Plouffe, " An Asynchronous Programming Language and Computing Machine", TR114a, Dept. of Information and Computer Science, University of California, Irvine, Dec. 1978.

[3] Arvind and R.E.Thomas, "I-Structures: An Efficient Data Type for Functional Languages", TM-178, Laboratory for Computer Science, MIT, Sept. 1981.

[4] Clark,K.L. and S.Gregory, " A Relational Language for Parallel Programming", Research Report of Imperial College of Science and Technology, DOC 81/16, Jul. 1981.

[5] Clark,K.L. and S.A.Ternland, "A First Order Theory of Data and Programs", IFIP 77, North-Holland Publishing, 1977.

[6] Cohen,J., "Garbage Collection of Linked Data Structures", Computing Surveys, Vol.13, No.3, Sep. 1981.

[7] Conery,J.S. and D.Kibler, "Parallel Interpretation of Logic Programming", Proc. of Conf. on Functional Programming and Computer Architecture, ACM, Oct. 1981.

[8] Gurd,J.R. and I.Watson, "Data Driven System for High Speed Parallel Computing", Computer Design, Jul. 1980.

[9] Ito,N., R.Onai, K.Masuda and H.Shimizu, "Prolog Machine Based on the Data Flow Mechanism", TM-016, Institute for New Generation Computer Technology, Tokyo, Japan, May, 1983.

[10] Ito,N., K.Masuda and H.Shimizu, " Parallel Prolog Machine Based on the Data Flow Model", Technical Report, Institute for New Generation Computer Technology, Tokyo, Japan (in Preparation).

[11] Masuda,K., N.Ito and H.Shimizu, " Simulation of a Data-Flow-Based Parallel Prolog Machine", Proc. of 27th National Conference of Information Processing Society of Japan, Nagoya, Japan, Dec. 1983 (in Japanese).

[12] Tanaka,H. et al.,"The Preliminary Research on the Data Flow Machine and Data Base Machine as the Basic Architecture of Fifth Generation Computer Systems", Proc. of International Conference on Fifth Generation System, Japan Information Processing Development Center, Tokyo, Japan, Oct. 1981.

[13] Kowalski,R., "Predicate Logic as Programming Language", IFIP 74, North-Holland Publishing, 1974.

[14] Shapiro,E.Y., "A Subset of Concurrent Prolog and its Interpreter", TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Jan. 1983.