ICOT Technical Report: TR-105

TR-105

Architecture of a Reduction-Based Parallel Inference Machine: PIM-R

by

R. Onai, M. Aso, H. Simizu,
K. Masuda and A. Matsumoto

May, 1985

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

Architecture of
a Reduction-Based Parallel Inference Machine : PIM-R

Rikio ONAI, Moritoshi ASO, Hajime SHIMIZU,
Kanae MASUDA and Akira MATSUMOTO

ICOT Research Center
Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F,4-28 Mita 1-chome,Minato-ku,Tokyo 108

ABSTRACT

This paper presents a highly parallel machine architecture for logic programs. A Reduction-Based Parallel Inference Machine : PIM-R is proposed, and the parallel execution mechanisms for PIM-R to run Prolog and Concurrent Prolog programs and software simulation results are described.

PIM-R uses the structure-copy method. It also uses the only reducible goal copy method, a unique process-structuring method, and the reverse compaction method to decrease the amount of copying and various copying-related operations and the number of packets passing through the network. PIM-R architecture features include the distributed shared memory for Concurrent Prolog, network nodes for efficient packet distribution, and the structure memory to store a part of structured data for reducing the copying overhead.

1. Introduction

ICOT is currently conducting research and development of knowledge processing software and hardware based on predicate logic languages. Because the basic operation of predicate logic is inference, the hardware is referred to as an inference machine and because it is inference-based, parallel processing is its fundamental mode of operation. In other words, an inference machine, or a parallel inference machine as we call it, is an "innovative" von-Neumann type machine rather than another von-Neumann machine based on sequential operation.

Currently there are several proposals for parallel inference machine architecture [Moto-oka 84], [Ito 84] and predicate logic languages [Shapiro 83],[Clark 84], [Pereira 84]. We have chosen Prolog and Concurrent Prolog [Shapiro 83] as the target languages of the machine. These have been selected by ICOT as the base languages for its Kernel Language version 1. We have selected a reduction-based approach which executes Prolog programs in OR-parallel and Concurrent Prolog programs in AND-parallel.

Let us briefly describe what caused us to develop the reduction-based machine. Assume that the expression "7+3" is to be reduced. The expression modifies itself using a rule about addition, to produce the result 10. The expression 10 is the answer, because it cannot further modify itself, i.e., it is irreducible. Thus reduction can be regarded as self-modification [Turner 79].

By contrast, the execution of Prolog or Concurrent Prolog programs generates resolvents from parent clauses (a goal and a clause); a resolvent obtained as an empty clause is the answer. This can be considered a process in which a goal modifies itself using a set of clauses, which are similar to rules. This close similarity between the execution of Prolog or Concurrent Prolog programs and the reduction process motivated us to research and develop a reduction-based parallel inference machine : PIM-R.

In PIM-R, if a process has multiple goals (the multiple goals, as a whole, are called the parent process), only the reducible goals, specified by various operators, are copied and reduced. Each resolvent generated contains a pointer to its parent process; the solution obtained is returned to the parent process using the pointer. That is, PIM-R executes Prolog and Concurrent Prolog programs by expanding and reducing a process tree. When the processing ends, the tree is logically deleted.

2. Parallel Execution of Prolog and Concurrent Prolog Programs

2.1 Parallel execution of Prolog programs

The parallel execution of Prolog programs is based on parallelism among arguments, AND-parallelism, and OR-parallelism.

An analysis of Prolog Programs showed that an average goal has two or three arguments [Onai 84]. Parallel execution based on the parallelism among arguments does not seem efficient, because arguments require consistency checking.

Prolog programs can generate multiple solutions. In that case, AND-parallel execution requires complicated consistency checking of arguments shared among goals in AND-relations, which reduces the merit of AND-parallel execution. For this reason, PIM-R does not use the AND-parallel execution either.

In contrast, parallel execution based on OR-parallelism is expected to perform highly efficient parallel processing on some problems (e.g., Bottom Up Parser, etc.), and hence has been chosen as the parallel execution mechanism of Prolog programs ("cut," "assert," and "retract" are excluded) on PIM-R. Thus, we refer to Prolog as OR-parallel Prolog in this paper.

This subsection discusses OR-parallel and AND-sequential execution of Prolog programs on PIM-R. Assume the following goal sequence and clause group:

```
Goal sequence  p1, p2, p3
Clause group
        p1 :- q1, r1, s1.
        p1 :- q2, r2, s2.
              :
              :
        p1 :- qm, rm, sm.
```

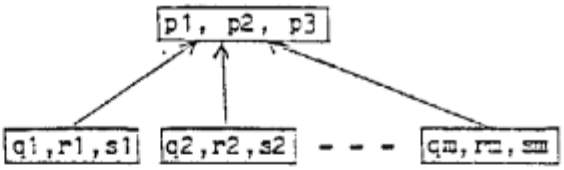


Fig. 2.1

This goal sequence is executed sequentially from left to right. Since only p1 is reducible, p1 alone rather than the entire goal sequence is copied and sent to the Unification Unit (discussed later) for unification (only reducible goal copy method). This only reducible goal copy decreases the amount of copying and shortens the packet length on networks. The unification generates m resolvents (i.e., child processes) with the goal sequence "p1, p2, p3" as their parent process (Fig. 2.1). The internal structure of a process is described in

Subsection 3.1.2.

Each child process has a pointer indicating its parent process to return a solution it obtains to that parent process. However, there is no pointer from a parent to a child. The parent process has a counter to store the OR-fork count, or the number of its child process. Child processes are called sibling processes because of their relationship. They are distributed among different processing units, (called Inference Modules), for OR-parallel execution.

Within a child process, "q1, r1, s1", for example, the leftmost goal q1 is reducible so q1 alone is copied (only reducible goal copy method) and transferred to the Unification Unit for unification.

2.2 Parallel execution of Concurrent Prolog programs

A Concurrent Prolog clause has the following format: h:-g|b.

Where, "g" is a goal sequence called a guard part, "b" is also a goal sequence called a body part, and the symbol "|" is called guard bar, or commit operator.

Once a goal is specified, the corresponding clauses (OR-related to each other) are searched in parallel. When unification succeeds on the guard part of a clause, the clause is selected, the binding environment obtained in the processing of the guard goals is opened up (commit operation), and the body part of that clause becomes a resolvent.

A goal in Concurrent Prolog can generate a single solution. If the guard part of any other clause succeeds in unification later, that clause will be deleted. In this sense, the guard bar functions as a cut symbol.

Two operators are used to specify the execution of goal sequences: parallel AND ",", and sequential AND "&". Goals connected with parallel AND-operators are executed in parallel. When goals connected with parallel-ANDs share a logical variable, they are in a "producer-consumer" relation and that variable is used for inter-goal communications. Thus it can be said that parallel ANDs use logical variables to implement inter-goal communications. PIM-R has a distributed shared memory called Message Board to store shared variables used in inter-goal communications.

In a goal sequence, "p1, p2" (",", is the parallel AND-operator), for example, p1 and p2 fork as different processes (this is called AND-fork), and unifications on the processes are performed in parallel. Consumers must wait for their variable to be assigned a value.

A shared variable can have a read-only annotation "?" attached (e.g., X?). A process which has a shared variable with a read-only annotation added (consumer process) cannot instantiate that variable. It must suspend until another process (producer process) instantiates the variable, or sends a message.

Assume the following goal and clause group:

```
Goal           p1
Clause group   p1:-g1|b1.
               p1:-g2|b2.
                   :
                   :
               p1:-gm|bm.
```

Fig. 2.2

Unification on P1 produces m OR-relation resolvents (children) (Fig. 2.2). Unlike the parallel processing method for Prolog programs which distributes m independent resolvents among different processing units, the parallel processing method for Concurrent Prolog passes them to the processing unit storing the parent p1 to first perform unification on their guard parts.

The parent p1 and children g1|b1, g2|b2, --- , gm|bm form a process (described in 3.1.2). When unification succeeds on the guard part of a child, that child asks the parent (p1 in the example above) whether it is the first child having succeeded in unification on its guard. The parent process has a commit tag called C-tag (described in 3.1.2 [1]) which is turned on by the child which first succeeds in unification. If the C-tag is already on when a child asks the parent, that child turns out not to be the first successful process and then enters the dead state.

In PIM-R, when a child successfully turns on the C-tag and performs the commit operation, the parent does not kill the child's siblings; rather any other sibling will commit suicide when it later succeeds in unification. We have chosen this approach because :

(1) The guard part is rarely so deeply nested as to cause a child which succeeds in unification after the C-tag is turned on to consume a huge amount of CPU time, and

(2) (1) means that it is not reasonable to perform the kill operation, which requires intricate control to prevent killing messages sent by the parent to its child from crossing any report on successful unification from children.

As described above, a sibling always checks the C-tag of its parent before it performs the commit operation. If the parent and children are assigned to different processing units, network traffic among the processing units occurs every time a child checks the C-tag. Since each Concurrent Prolog clause has a commit operator, the network traffic would become enormous. Therefore executing child processes on different processing units in an OR-parallel manner will not reduce the time for problem solving; in some cases it could increase the problem solving time.

To prevent this situation, parallel execution of Concurrent Prolog on PIM-R does not distribute siblings among different processing units, but passes them to the unit storing the parent for guard processing. On the other hand, goals connected with parallel AND-operators are distributed to different units as new processes and executed in parallel.

To sum up, the PIM-R executes Prolog programs in OR-parallel and Concurrent Prolog programs (goals are connected with parallel AND-operators) in AND-parallel.

3. PIM-R Architecture

As shown in Fig.3.1-1, PIM-R basically consists of two types of modules, an Inference Module and a Structure Memory Module, besides networks connecting these modules.

(Fig. 3.1-1 is inserted.)

3.1 Inference Module

The Inference Module (IM) consists of two units: the Unification Unit (UU) and the Process Pool Unit (PPU) (Fig.3.1-2).

(Fig. 3.1-2 is inserted.)

3.1.1 Unification Unit (UU)

(1) Clause Pool (CP)

The Clause Pool (CP) in each IM stores the same clauses. Each word in CP is 32 bits, the most significant 8 bits indicate the data type.

PIM-R uses the structure-copy method to increase the independence of individual processes and decrease the network traffic due to structure sharing. On the other hand, copy overhead due to the use of the structure-copy method increases. This section describes the internal format of a clause which is used to prevent copy overhead as much as possible. Appendix. 1 contains explanations of the data types in the internal format.

The Clause Pool consists of a Clause Definition Group Management Block and a Clause Definition Block.

The Clause Definition Group Management Block stores the number of clauses in OR-relation, a pointer to the Clause Definition Block where each clause is stored, and the data type of the first argument of the head literal of a clause. This last information is required to allow the Matcher to select unifiable clauses.

The Clause Definition Block stores the definition of a clause and consists of a header, a variable area, a literal header, a literal area, and a structure area (Fig.3.1-3).

The header contains the clause length, the head address of the structure area, and the head address of the literal header. The structure area head address is assumed to be stored at address 0 and other addresses are relative to address 0. When the result of reduction is passed as a packet, packet length and pointer to a parent are inserted before the word of the structure area head address. Since addresses in a Clause Definition Block are relative, address changes are not necessary.

The variable area contains the number of variables and binding information on individual variables.

The literal header holds the literal count (at first, the number of body literals in sequential AND-relation + 1), and the head literal; it also stores body literals in sequential AND-relations (a commit operator is regarded as a literal) in descending order. Literals with no arguments are stored in the literal header, while literals with one or more arguments or those connected by parallel AND-operators are held in the literal area in descending order.

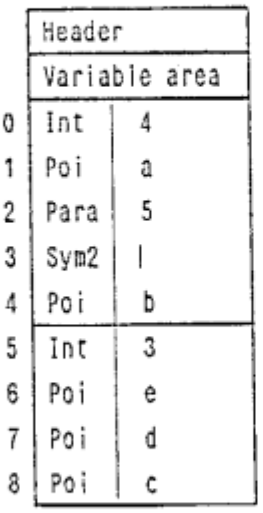
A literal stored in the literal area contains the argument number + 1, a pointer to the Clause Definition Group Management Block, and arguments. Storing literals in descending order in the literal header and the literal area easily supports reverse compaction. This is explained in Subsection 3.1.2 [1] (2) using an example.

Any structured data (information on "List", "Vector", and "Channel Information") is stored in the structure area. If there is no structured data, the structure area does not exist.

(Fig.3.1-3 is inserted.)

Let us explain parallel and sequential Basically the guard part, commit operator "|", and body part of a Concurrent Prolog clause are considered to be in sequential AND-relation. Therefore parallel and sequential AND-descriptors are used only when the parallel AND-operator "," appears in the guard and/or body parts, or the sequential AND-operator "&" appears in any goal connected by parallel AND-operators.

[ex.3.1.1-1] An example clause is a :- b | c, d, e.



As shown in left figure of Clause Definition Block of the above clause, the parallel AND-descriptor "Para" indicates a set of body goals in parallel AND-relation, i.e., connected by ",".

The area pointed to by Para contains the number of literals in parallel AND-relation and the individual literals,

which are arranged in descending order for reverse compaction.

As shown by this example, the commit operator "|" and the literals b, c, d, and e are considered to be in sequential AND-relation.

(2) Matcher and Unifier

The Matcher chooses unifiable clauses according to the data type of the first argument of the goal passed from the Parent Process Unit. When the goal is not a re-try goal or a built-in predicate, the Matcher sets an OR-candidate count field in the OR-fork counter to the count of the chosen clauses, i.e., candidate clauses. There are two fields in the OR-fork counter : an OR-candidate count field and a return count field. The return count is initialized to 0. It then sends the goal and the location of the candidate clauses to the Unifier.

The Unifier stores the goal sent from the Matcher in the goal memory and a clause copied from the CP in the clause memory, unifies the goal with the clause, generates the final result in the goal memory, and passes it to the output buffer. Also the Unifier executes built-in predicates.

When unifications of a goal and candidate clauses are finished, the return count (equal to the unification success count) is returned to the parent process as an OR-fork count.

When unification with a unit clause succeeds in OR-parallel Prolog processing, the unification result is returned to the parent process in the Process Pool (PP). When unification with a clause other than a unit clause succeeds, however, the new

resolvents are returned to the PP in the same IM or distributed via a network among the other IMs according to a specified distribution strategy.

By contrast, in Concurrent Prolog processing the unification result is always returned to the PP in the same IM. When unification suspends, information for future re-try processing (ex. the given goal, the location storing the clause on which the unification was attempted, etc.) is returned to the PP in the same IM and a Suspend Process Control Block (discussed later) is produced.

During unification variable area or structure area may increase. Since each address is represented as the displacement from the head address of the literal area or head address of the structure area, the Unifier changes literal area and structure area head addresses only if necessary. This reduces unification processing.

3.1.2 Process Pool Unit (PPU)

The Process Pool Unit (PPU) consists of two types of memories (Process Pool and Message Board) and two types of controllers (Process Pool Controller and Message Board Controller).

[1] Process Pool and Process Pool Controller

(1) Process Pool (PP)

The Process Pool (PP) is a memory for storing processes (32 bits/word, same as the Clause Pool). PIM-R employs a process configuration method capable of minimizing communications between IMs.

A process consists of Process Control Blocks (PCB) for storing control information of a goal sequence, a Process Life Block (PLB) to manage the number of Process Control Blocks, and Process Template Blocks (PTB; a PTB and a PCB makes a pair) to hold the template of a goal sequence. These blocks are assigned to an IM as a set.

Simply put, a reducible goal in a PTB is sent to the UU, and when its solution is returned to a PCB, the goal sequence in the corresponding PTB is copied, the binding environment is assigned, and a new goal sequence (a PCB-PTB pair) is generated under the same PLB. The following is a description of these blocks.

(a) Process Life Block (PLB)

The PLB is the top level block in a process and contains the commit tag, the number of PCBs under the PLB, and other information. At Concurrent Prolog execution, the commit tag is turned on by the PCB in the process that first succeeds in executing its guard part.

(b) Process Control Block (PCB)

The PCB contains the state of a goal sequence (reducible (ready), run (unification is under way), wait (waiting for a solution to be sent from a child process), dead, or suspend (consumer process is waiting for a shared variable to be

connected with a value)), reduction level, number of OR-forks (when an OR-parallel Prolog program is executed) or AND-forks (when a Concurrent Prolog program is executed), number of returns (i.e., return from forked processes), a pointer used to connect to the Ready Process Queue, and other information. The reduction level means the relative depth of each process, assuming that the depth of the process which corresponds to the root of the process tree is 1.

When the goal sequence state is suspend, the PCB is referred to by the special term Suspend Process Control Block (SPCB). The difference is that the PTB address of the channel causing the suspend state is stored in the PCB, and that the suspended goal is stored in the PTB under this SPCB.

(c) Process Template block (PTB)

When under a PCB, a PTB has the same internal format as clauses in the Clause Pool (CP) except for the clause length. When under a SPCB, by contrast, it stores the suspended goal and the CP address of the clause with which unification of the suspended goal was attempted.

(2) Process Pool Controller (PPC)

(a) Process creation, renewal, and deletion

(a-1) Process creation

When a new resolvent is returned from the UU, a new process which consists of a PLB and a PCB-PTB pair is created.

(a-2) Process renewal

Process renewal is the updating of the fork count and return count and the creation of a new PCB-PTB pair.

When the UU returns the OR-fork count (indicating successful unification), the fork count is incremented by that OR-fork count in Prolog program execution. In Concurrent Prolog, goals in AND-relation are generated as different processes. Therefore, when goals connected with parallel AND-operators are encountered in a clause, the AND-fork count is set to the fork count in the PCB, and these AND-related goals are distributed to the IM concerned or to another IM according to a distribution strategy.

When a new PTB is created from an old PTB, reverse compaction is executed. The following example illustrates process renewal processing and reverse compaction precisely.

(Ex.3.1-1) Process renewal and reverse compaction in the PP

The Clause Definition Block of the clause "p(1,[X|Y]):-q(1,X)&r(X,Y)." is shown in Figure 3.1-4; "&" is a sequential AND-operator.

If unification succeeds, a Clause Definition Block like this is passed to the Process Pool to form the PTB of the process. The PTB is shown on the left in Figure 3.1-5.

In the PTB, one word of structure area head address (the second word) is the address 0.

The storage allocation of a literal is determined by the displacement relative to the head address of the literal header. Also the position of a structured data value is indicated by the displacement from the head address of the structure area. The head address of the variable area is always stored in the fourth word.

The literal count is stored in the first word of the literal area. The literal count address plus the literal count indicates the word containing the pointer to a goal literal in the reducible or run state. At first, the literal count is the body literal number plus 1, that is, 3. Since there is Lit 12 at the position of displacement 3 to the literal count address, a goal q(1,X) is reducible. Therefore only q(1,X) is copied and sent to UU.

We assume that q(1,X) succeeds in unification with a unit clause and the result q(1,2) is returned to the parent process in the PP. (As described above, a new process is not generated, since a goal is unified with a unit clause.) In Prolog programs when unification succeeds and a binding environment is returned, multiple solutions may be returned. Therefore, the entire PTB must be copied before the binding environment is written into the block. After copying, the new binding environment X=2 is stored in the new PTB.

To improve the efficiency of use of the Process Pool, reverse compaction is performed on the PTB by removing the q portion, moving the structure area upward and decreasing the literal count by one. At compaction since literals are stored in reverse order, it is sufficient that the last literal area, the q portion, be removed.

This may change the locations at which each variable or structured data is stored. But only the structured data area head address in the second word needs to be changed. Since the storage locations of structured data are indicated by displacement from the structure area head address, pointers to the structured data need not be changed. If the variable area length changes, the literal header head address and structure area header address must also be changed. The new literal count indicates "Lit 8" which points to the next reducible goal r(2,Y). The right side of Fig.3.1-5 shows the PTB after the above operations and compaction. When the structure area gets longer, that is, new structured data is added, this new structured data is stored in the bottom of the structure area.

By contrast, in the execution of Concurrent Prolog and of built-in predicates, the PTB need not be copied when a solution is returned, because a goal produces only one solution (i.e., only one child can survive). If clause length does not get longer at reverse compaction, direct overwriting to the process template is possible. This method reduces the amount of copying and the occupied space in the Process Pool.

(Fig.3.1-4 is inserted.)

(Fig.3.1-5 is inserted.)

(a-3) Process deletion

When a PCB enters the dead state (i.e., fork count = return count), the PPC sets the PCB state dead and increments the PCB return count in the corresponding PLB by 1.

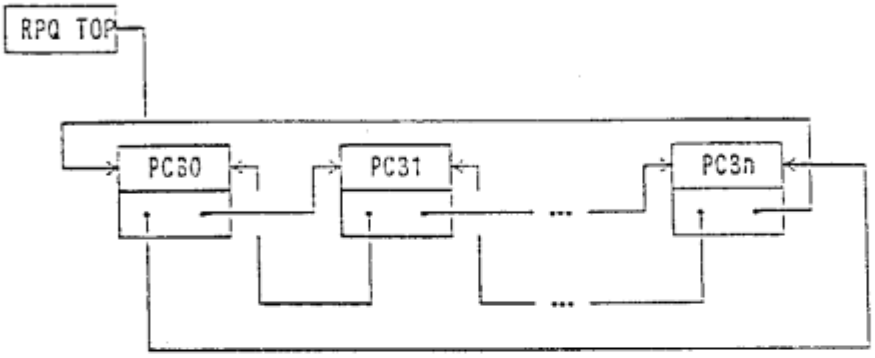
To sum up, the introduction of the PLB into a process can make it unnecessary to report each generation or deletion of a goal sequence to its parent process, irrespective of how many goal sequences are generated or deleted in that process. Therefore the use of such process-structuring methods permits a decrease in the number of packets passing through the inter-IM network. If the memory has sufficient space, the solutions can be obtained in less time by halting the dead process handling routine in the PPC and stopping the generation of fork down packets (i.e., decreasing network traffic).

In Appendix 2, we explain the inter-relationship among the PLB and PCBs.

(b) Local program-executing strategies using the reduction level

PIM-R is expected to have 100 or more interconnected IMs. If such a system has a general scheduler, it is difficult for the general scheduler to control the entire execution strategy. Therefore, the program execution control in PIM-R is only performed locally within a single IM.

As shown in the figure below, PCBs which hold the control information of a goal sequence including any reducible goals, called Ready PCBs, are linked to the Ready Process Queue (RPQ) in the IM.



When a program execution strategy is not particularly specified, the PPC attaches new Ready PCBs to the tail of the RPQ, and sends the head Ready PCB to the UU. Some programs, however, require only one of multiple solutions to be obtained. The reduction level is used in a strategy to execute such programs. A pseudo-depth-first reduction strategy can be implemented in the IM, by placing a PCB with higher reduction level (i.e., a PCB which is farther away from the process tree root, or positioned at a deeper level) at a position closer to the head of the RPQ. Of course a pseudo-breadth-first reduction strategy can also be implemented in the IM by placing a PCB with a lower reduction level (i.e., a PCB nearer to the process tree root, or positioned at a shallower level) at a position closer to the head of the RPQ. Thus, the reduction level allows the control of local reduction strategies in the IM.

(c) Garbage Collection

Except for instantiated long structured data, data is copied eagerly. Therefore, such data is not shared among PCB-PTB pairs (goal in a process). When a PCB-PTB pair dies, data relevant to the PCB-PTB pair becomes garbage. It is easy for the PPC to decide whether a PCB-PTB pair is garbage or not by checking

counters relevant to the child process number. If there is no child process, that process is dead. This check is only light processing.

[2] Message Board (MB) and Message Board Controller (MBC)

Each IM has a distributed shared memory called Message Board (MB) to store channel variables in Concurrent Prolog programs. The Message Board Controller is designed to reduce the load of PPC processing.

(1) Message Board (MB)

As stated above, PIM-R separates variables used as channels (these variables are called simply channels) from the other variables, and stores channels on the MB.

It is assumed that goal sequence p1(X), p2(X?) becomes reducible, p1(X) and p2(X?) become different processes as the result of AND-fork processing, and these processes are distributed into different IMs. Since there is an MB in each IM, the cell for channel X is stored in the MB of either IM and process p1(X) and p2(x?) share the cell. If the cell of channel X is stored in the MB of the IM which stores process p1(X), process p2(X?) reads a value from it through the network.

The characteristics of a channel are dynamically inherited by another variable at execution. For example. when unified with a channel in a goal, an argument variable in a clause becomes a channel.

The MB consists of channel cells, value cells, and a suspend process list.

Channel cells consist of four words and contain a write tag, a suspend tag, a pointer to a value cell, the suspend process count, and the head address of the suspend process list.

A value cell stores a value sent from a producer process. The internal format of a value cell is the same as that of the Clause Pool.

Suspend Process List stores a pointer to a suspend process. Suspend processes themselves are held in the Process Pool.

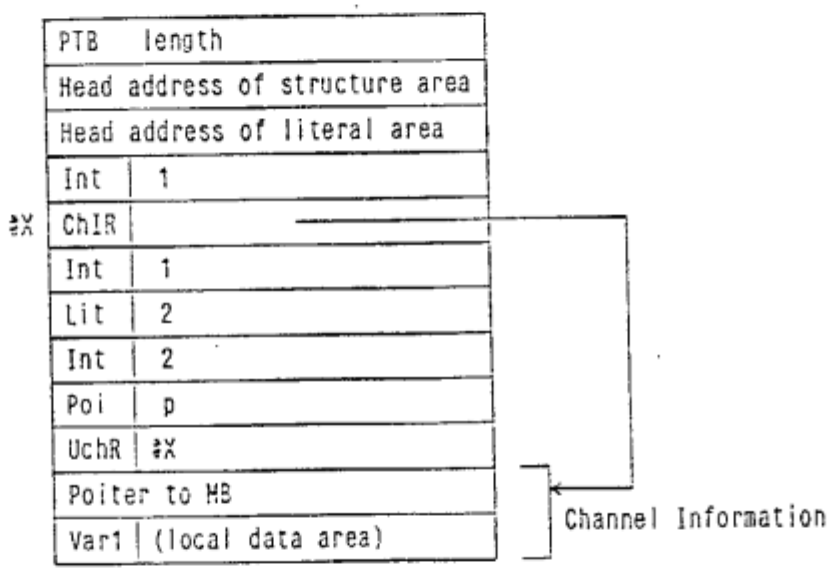
(2) Message Board Controller

In Concurrent Prolog, when a consumer process suspends the PPC requires the MBC to check whether a producer process has already sent a message to that MB containing the cell of the channel causing the suspension. If a message has arrived, the MBC sends the message to the PPC to activate the consumer process; otherwise, the MBC writes the PP address of the consumer process in the Suspend Process List.

If the MB containing the channel cell is in another IM when a consumer process suspends, a channel value read packet is sent via a network to that MBC. If a message has already arrived at that MB, an activation packet containing the message is returned to the consumer process as a packet. When a producer process sends a message, i.e., connects a channel with a value, that message (value) is written in a value cell on the MB. If at this time the Suspend Process List contains any suspend consumer

process, the message is sent to that process. If any suspend process exists in the PP in another PRU, an activation packet is transferred via a network as a packet.

A channel can be described with two "channel information" words stored in the structure area in a Process Template Block (PTB). Given a goal sequence " p(X), c(X?) " , p(X), which is placed in the PP in an IM as a result of an AND-fork, has a PTB as follows:



(ChIR is a data type which stores a pointer to channel information.)

The first word of Channel Information is a pointer to an appropriate channel on the MB or the channel information in the parent process. The second word stores local data. When unification on a guard succeeds and the commit operation is performed, this local data is written into the appropriate channel cell on the MB or the local data area in the channel information of the parent process. This is explained by examples in Appendix 3.

3.2 Structure Memory Module

Efficient processing of structured data is very important in the design of a parallel inference machine. In general, many data items manipulated in Prolog applications include complex and large structured data. For manipulating this large structured data, however , there is a problem of large overhead caused by data copying in a structure-copy system. A function for sharing structured data is required.

We have introduced the structure memory concept to reduce the copying overhead. Our Structure Memory Module stores only a large structured data, such as large combined lists and vectors, although the structure memory in a dataflow-based parallel inference machine stores all of the structured data [Ito 84].

As shown in Fig.3.1-1, each Structure Memory Module(SMM) is connected to several Inference Modules (IM) via the IM-SMM Network and a part of the structured data is accessed by the Unification Unit (UU) in the IM on a unification-demand basis. Sharing structured data among several IMs involves frequent access concentrations and conflictions on a particular SMM. In order to avoid this problem, we distribute several SMMs holding the same memory image over the system and enable concurrent multiple reading.

(1) Sharing method

The structure area in each PTB includes all of the structured data with the undefined variables and ground instances. Different kinds of sharing levels are available, such

as sharing literals, sharing the entire structured data, and sharing ground instances. If we choose a sharing method which shares the structured data with undefined variables, it will be required to copy the structure area when an undefined variable of the structure area is bound to some value. In order to avoid this copying overhead, we have chosen the most basic method : that of sharing only ground instances which do not include any undefined variables. As the first step, in compiling a Prolog program, we divide structured data in a clause into one part including the undefined variables and one part for the remainder. As the remainder does not include any undefined variables, it will not undergo any more change. Such large structured data is a candidate to be stored in the SMM as a ground instance. Fig.3.2-1 shows an example of sharing ground instances.

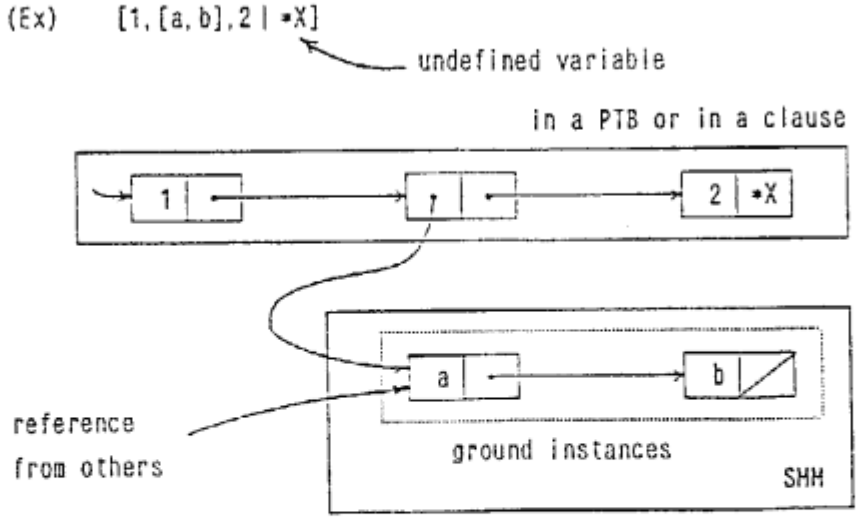


Fig. 3.2-1 Example of sharing ground instances

This sharing method can maintain highly parallel environments among processes, because only reading operations are executed and the shared data consists of ground instances only requiring no rewriting [Moto-oka 84]. At present as the first step, programmers specify the many ground instances from structured data in a clause and these are stored in the SMM. This sharing method places a pointer in the clause which might be also referenced by a new process, because that pointer is transferred to the new resolvent at unification, causing sharing of the ground instances in the SMM. As a result, clauses and goals have the pointers instead of large structured data, thereby reducing the size of clauses, goals, and network packets.

(2) Combined representation of structured data

A list has a construct operator as its functor and two arguments (car and cdr), and only these two arguments are stored in the SMM. A vector is composed of its size and one or more elements, and these are all stored in the SMM. In general, these structured data are stored by either the list-based representation with pointer areas or the record-based representation storing data in successive memory cells.

The list-based scheme needs more memory space but allows list data to be fetched by tracing one pointer at a time, thus featuring the ability to easily retrieve data using pointers. On the other hand, the record-based scheme has problems of overhead due to data reading via an address table and difficulties in implementing fast, successive read operations. However, it can increase utilization efficiency for the storage of large

structured data like vectors.

The present plan uses both schemes and separates the memory for lists from that for vectors. As shown in Fig.3.2-2, lists are stored in the List Data Memory in a list format, while vectors are stored in the Vector Data Memory in a record format and accessed via a Vector Address Table(VAT). The access to the SMM occurs on a small amount basis; we call this a block access. A list block consisting of car and cdr parts is fetched to the UU by a list read request and a vector block is fetched by a vector read request respectively.

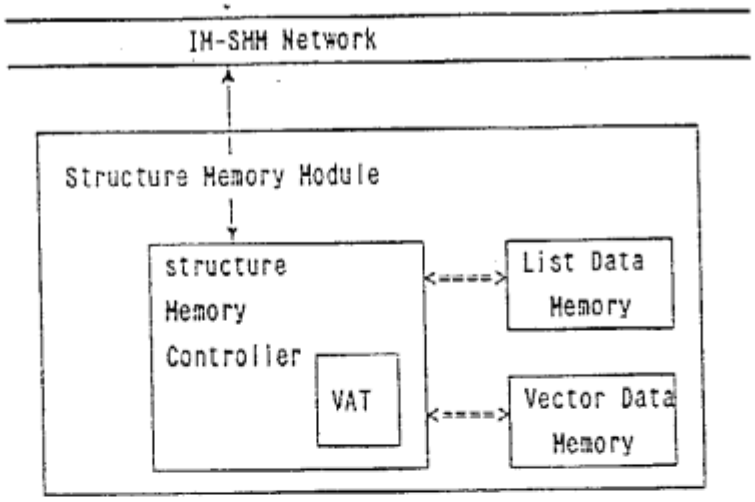


Fig. 3.2-2 Structure Memory Module configuration

(3) Lazy unification between arguments

When unification requiring the structured data stored in the SMM is executed, the SMM is accessed. In this case, PIM-R uses a lazy unification between arguments to avoid unnecessary unifications. If there is an argument which refers to the SMM in either the reducible goal or the unifiable clause and if the other argument for unification is neither an undefined variable nor an atom, then this unification will be delayed and the other unifications between arguments not refering to the SMM will be executed with higher priorities. When all these unifications succeed, the delayed unification refering to the SMM will be executed.

Our method stores at compilation only ground instances of a program in the SMM and does not execute, at present, dynamic memory write operations for the SMM. The structured data in the SMM are referenced by at least one clause and will become garbage when all the solutions associated with a query are obtained. We are now developing a detailed software simulation to evaluate the effectiveness of our structure memory method. We plan to measure the effective size of ground instances stored in the SMM, the number of memory access operations, and so on.

3.3 Network

At execution of Prolog or Concurrent Prolog programs on PIM-R, when a goal succeeds in unification with a rule, a child process (new resolvent) is generated. However, even if a goal succeeds in unification with a fact, the result is only returned to the PCB in a parent process and a child process is not generated. [Onai 84] shows that the average OR-relation number is about 2 or 3 in Prolog programs that consist mainly of rules. Therefore, it is possible to map an average process tree onto a cyclic mesh structure.

In programs of logic languages such as Prolog, when there are several solutions, location (depth) of each solution in the AND-OR tree are usually different and children forked simultaneously rarely return solutions to a parent process at the same time. Thus, the mesh-type network node, onto which the upper part of a process tree is mapped, only rarely gets overloaded by concentration of solutions returned to a parent process.

The access length of network nodes at execution on PIM-R is about 1, since most of accesses are between neighbor nodes. (Accesses to the Message Board are sometimes not between neighbor nodes.)

For these reasons, we chose mesh type structure for the inter-IM network. (Figure 3.1-1) In the inter-IM network, nodes are located at mesh points and an IM is connected to each node.

Network nodes can dynamically control the distribution of a child process to each IM using such information as the input buffer length of the Packet Switch in a PRU. Traffic in networks is bi-directional. Each network node can be configured using microcomputers with communication functions and a high-speed memory, such as Transputers [Inmos 84-b] (four channels, each with a transfer rate of 10 Mbs and 4 Kbyte static RAM with a cycle time of 50 nsec).

The network between IMs and the SMM, or the IM-SMM network, is of the equal distance type and at present is expected to be implemented with a shared bus.

4. Software simulation

The simulator for PIM-R is written in Prolog, and runs on DEC2060 or VAX-11/780 or VAX-11/785.

4.1 Simulation conditions

Simulation is performed under the following conditions:

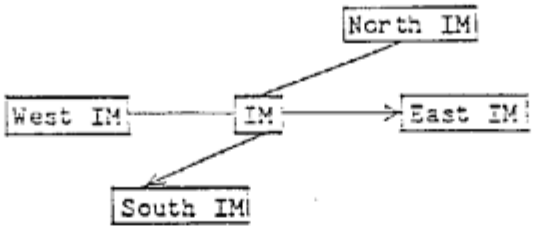
(1) A network is ideal, excluding packet conflict.

(2) Each unit has a buffer of sufficient size.

(3) The performance of PRUs (process creation, renewal, and deletion) and UUs is determined by the number of steps required when they are written in an ordinary assembly language. [Moto-oka 85] who adopted the all goals copy method, shows that the average time of a unification and size reduction in a 6-Queens program is about 42 μsec.

Since PIM-R uses the only reducible goal copy method and the amount of copying for this method is less than that for the all goals copy method, we assume that UU in PIM-R is able to execute a unification and result making in 100 μsec at least. (we think that UU is able to execute those operations in under 50 μsec if special hardware for the UU is implemented.) Since the average packet length through networks of 4-Queens is about 20 words (about 600 bits), we assume that network speed is 10 Mbs and average network delay is 60 μsec.

(4) A new process is distributed to an IM at child process creation in Prolog programs and at AND-fork in Concurrent Prolog programs according to distribution strategy. This simulation adopts the following static and cyclic distribution strategy: IM concerned, East IM, South IM, IM concerned,...



(Fig.4-1 is inserted.)

4.2 Simulation results

4.2.1 Prolog program

The 4-Queens program was run to collect the necessary data.

(1) Effect of number of Inference Modules (Figure 4-1)

The 4-Queens program increases in performance as the number

of Inference Modules (IM) increases. Processing time decreases as more Inference Modules are used up to seven or eight units. Then it levels off. This trends roughly corresponds to the average level of OR-parallelism, about 6, resulting from a dynamic analysis [Onai 84]. The results show that PIM-R is able to execute parallel processing corresponding to the parallelism in the Prolog program.

(2) Halting dead child processing

The number of packets passing through the network is shown by their types in Table 4-1.

Table 4-1

	Two IMs	Four IMs
Total number of packets	151	207
Number of true return packets	31	47
Number of OR-fork packets	60	80
Fork down packets	60	80

The fork-down packet is used by a child process to inform its parent process that it has entered the dead state. It is concerned with garbage collection, and is not directly related to the solution-obtaining processing. Therefore, if the Process Pool has sufficient space, lower priorities can be given to dead child processing and the generation and transfer of fork-down packets in the PPC. Higher priorities are given to first executing processing required to obtain solutions and to transfer the respective packets. This can reduce the number of packets passing through the network by about 40% for two and four IMs. Also it alleviates processing load in the PPC, resulting in about 8% and 15% reductions in processing time to obtain the first and second solutions for four IMs respectively.

(3) Local execution control using reduction level

Local pseudo depth-first execution was tried giving higher priorities to Ready PCBs with deeper reduction levels and to the processing of true-return packets from child processes. At the same time, lower priority is given to fork-down packet processing. The result was that 12% and 17% reductions in processing time to obtain the first and second solutions for four IMs respectively were achieved.

(4) Network load average and network speed

The load average of each link is about 10% for two IMs and 6% for four IMs. (There are 2 links for two IMs and 8 links for four IMs.) The more IMs the less network load average. Simulation condition (2) is barely appropriate.

The maximum input buffer length of the Packet Switch is 15 for two IMs and 14 for four IMs.

If we speed up the network from 10Mbs to 60Mbs, we get about 10% reduction in processing time for four IMs.

(5) PPU load average and dynamic process distribution

When 60Mbs network speed for four IMs is achieved, the PPC in each IM recorded load averages of 42%, 55%, 54%, and 80%. Relatively wide discrepancies exist among these figures, because the child process distributing strategy at OR-fork was fixed. In this case, the average input buffer lengths of the packet switches are 0.46, 0.69, 0.49, and 4.5 units. These results correlate with the load averages of the PPC.

When network nodes are used to dynamically distribute child processes to the IM whose packet switch has the shortest input buffer length, the processing time necessary for obtaining the first and second solutions can be decreased further by 10% and 14% for four IMs; then the PPC in each IM has balanced load averages of 69%, 72%, 62%, and 65%.

4.2.2 Concurrent Prolog program

The Quicksort (ten elements) program was run to collect various data items.

(1) Effect of number of Inference Modules (Figure 4-1)

The Quicksort program increases in performance as the number of IMs increases. Processing time decreases as more Inference Modules are used up to five or six units. Then it levels off. Since this example has a parallelism of about 4, the results show that PIM-R is able to execute parallel processing corresponding to the parallelism in the Concurrent Prolog program.

(2) Halting dead child processing

Table 4-2 shows the number of reductions and other data at the execution of Quicksort.

Table 4-2

Number of reductions	284
Number of successes	196
Number of failures	19
Number of suspended reductions	69

Table 4-3 shows the number of packets passing through the network during Quicksort execution by their types.

Table 4-3

	Two IMs	Four IMs
Total number of packets	141	211
Number of true packets	17	17
Number of AND-fork packets	21	26
Number of fork down packets	21	26
Number of MB-related packets	82	140

These tables show that, while 284 reductions occurred and 196 of them succeeded, 141 packets passed through the network for two IMs and 211 packets for four IMs. In other words, a packet passed through the network each time about 1.3 or 2 reductions occurred. Unlike the Prolog program, the MB-related packets account for 58% of the total for two IMs and 66% for four IMs, as shown in Table 4-3. This means that halting dead child processing and generation and transfer of fork-down packets, an effective approach in the Prolog program, could reduce the number of packets passing through the network by only 15% for two IMs and 12% for four IMs. Unlike the fork down packets, these MB-related packets cannot have a lower transfer priority attached; if they do, no solution will be obtained. Therefore, faster packet transfer is more critical for Concurrent Prolog.

(3) Effect of Message Board Controller (MBC)

A channel cell is allocated on the MB when unification succeeds in the UU, a new child process is returned to the PPU, and there is a new channel. At execution of Quicksort, 88 channels are stored on the MB and, as Table 4-2 shows, successful unifications total 196. This requires the cell for a channel to be allocated on the MB every time about 2.2 unifications succeed. Thus the speed of allocating a channel cell on the MB influences PIM-R processing speed. This problem can be eliminated by introducing a Message Board Controller (MBC) to handle MB-related

processing. If the MB-related processing were handled by the PPC, instead of the MBC, processing time would increase by 13% for one IM and by 10% for four IMs.

(4) AND-OR parallel execution

When Concurrent Prolog programs are executed in AND-OR parallel, child processes distributed in the different IMs from the parent process have to check the commit tag in the parent process through networks when guard execution is successful. Child processes have to wait for the return packet from the parent process. This increases network traffic. As the result, the processing time of AND-OR-parallel increases about 13% over that of AND-parallel in case of four IMs at execution of Quicksort. OR-parallel execution is not suitable for Concurrent Prolog programs.

5. Conclusion

This paper described the architecture of PIM-R, a reduction-based parallel inference machine, parallel execution schemes of Prolog and Concurrent Prolog programs on PIM-R, and software simulation results.

To decrease the amount of copying and miscellaneous copying-related operations due to the use of the structure-copy method and to minimize the number of packets passing through the network, PIM-R uses the only reducible goal copy and reverse compaction schemes as well as introducing the Process Life Block (PLB) into a process.

As for architecture, PIM-R uses a Message Board (MB), i.e., distributed shared memory for channels between AND-parallel processes, to handle Prolog and Concurrent Prolog programs equally. It was confirmed that the MB permits PIM-R to execute Concurrent Prolog functions including back communication and finite-length buffer communications. The PIM-R architecture is also characterized by the introduction of network nodes for efficient packet distribution, and by the use of structure memory for storing ground instances in large-scale structured data.

The simulation using PIM-R software simulators demonstrated that PIM-R is able to execute parallel processing corresponding to the parallelism in Prolog and Concurrent Prolog programs. In other words, the number of Inference Modules does affect the performance of parallel processing.

It was also confirmed that the dynamic distribution of child processes by network nodes, introduction of the Message Board Controller (MBC), halting of dead child processing, stopping the generation and transfer of fork-down packets, and local pseudo-depth-first execution using reduction level are all effective measures.

At Concurrent Prolog execution, over half the packets sent through networks are related to MB access. Therefore, we think that it is not sufficient to increase packet transfer speed. It is also necessary to decrease the relative number of MB-related packets to other packets. It is a way of introducing modularity into the language to enlarge grain size of AND-processing of Concurrent Prolog.

Channels in Concurrent Prolog are used for communication and synchronization among goals. Since a channel is a logical variable, a producer and a consumer have to execute unification for sending and receiving a message respectively. It causes a reduction in speed of message transfer. Since a value and next channel are usually passed through a channel, many channels have to be used for only one Occam-like channel [Inmos 84-a]. Also since channels in Concurrent Prolog are generated dynamically, garbage collection for channel cells, that is, for the Message Board has to adopt a complicated method like reference count method, that needs CPU time and causes network traffic. Clearly, research into the implementation problem of communication and synchronization in logic-type languages not using logical variables is necessary.

Though PIM-R now adopts 10 X 10 organization (Figure 3.1-1), we will investigate the hierarchical N X N organization of PIM-R and the clause allocation problem in Clause Pools.

At present we are developing a detailed software simulator written in Occam and a dedicated simulation system consisting of sixteen microcomputers. We plan to conduct various detailed simulations of many programs, including those for researching the effect of shared structured data, with these tools to validate and enhance PIM-R.

[References]

[Clark 84] Clark,K.L. and Gregory, S., "PARLOG:Parallel Programming in Logic", Research Report DOC 84/4, Dept. of Computing, Imperial College London, 1984.

[Inmos 84-a] Inmos Limited, "Occam Programming Manual", Prentice Hall International Series in Computer Science, 1984.

[Inmos 84-b] Inmos Limited, "IMS T424 Transputer Reference Manual", 1984.

[Ito 84] Ito, N. and Masuda, K., "Parallel Inference Machine Based on the Data Flow Model", Proc. of the International Workshop on High Level Computer Architecture 84, 1984.

[Moto-oka 84] Moto-oka, T., Tanaka,H. et al., "The Architecture of a Parallel Inference Engine-PIE- ", Proc. of Int. Conf. on Fifth Generation Computer Systems, 1984, ICOT, 1984.11.

[Omai 84] Omai,R., Shimizu,H. et.al., "Analysis of Sequential Prolog Programs", ICOT Technical Report TR-048, 1984

[Pereira 84] Pereira, L.M. and Nasr, R., "DELTA-PROLOG: A Distributed Logic Language", Proc. of Int. Conf. on Fifth Generation Computer Systems 1984, ICOT, 1984.

[Shapiro 83] Shapiro, E.Y., "A subset of Concurrent Prolog and Its Interpreter", ICOT Technical Report TR-003, 1983.

[Turner 79] Turner, D.A., "A New Implementation Technique for Applicative Languages" Software-Practice and Experience, No.1, vol.9, 1979.

[Appendix 1] Table of data types

Type Classification			Abbre-viation	Note
Type	SubType1	SubType2		
Variable	Void-variable		Void	VarR is the type of variables stored in the literal area or the structure area. It refers to Void or Var1 in the variable area. Var1 is the type of variables in the variable area. UChR, RChR and WChR are the types of channels stored in the literal area or the structure area ; these refer to ChIR in the variable area. ChIR refers to Channel Information in the structure area. A Undef Channel means that it cannot be determined at compile time whether it is a read channel or a write channel. Sym0 are built-in predicates such as "<" and ">" , which require no variable binding. Sym1 are built-in predicates such as "is" and "=" , which require variable binding. Sym2 are built-in predicates , such as "guard", "true", and "fail" processed by PPU. The type of Lit contains a pointer to a goal in the literal area. The type of Poi refers to Clause Pool or Process Pool. "Structured in SHM" is the type of structured data stored in SHM.
	Variable-1'st		Var1	
	Variable-Reference		VarR	
Channel	Undef Channel	1st	UCh1	
		ref.	UChR	
	Read Channel	1st	RCh1	
		ref.	RChR	
	Write Channel	1st	WCh1	
		ref.	WChR	
Atomic	User Defined Atom		Atom	
	Nil		Nil	
	System Symbol	Type0	Sym0	
		Type1	Sym1	
		Type2	Sym2	
	Integer		Int	
	Real		Real	
Structured	List		List	
	String		Strg	
	Vector		Vect	
	Channel Information Reference		ChIR	
	And	Parallel	Para	
		Sequential	Seq	
	Literal		Lit	
	Pointer		Poi	
Structured in SHM	Variable	void	SPVo	
		Var 1st	SPV1	
	Atomic		SPAt	
	Non Ground (not instantiated)	List	SPNL	
		String	SPNS	
		Vector	SPNV	
	Ground (instantiated)	List	SPGL	
		String	SPGS	
		Vector	SPGV	

[Appendix 2] Internal structure of a process and inter-relationships among processes

(Example A-2-1) For Prolog

programs, as follows:

```
?- go.

go :- a, b.

a :- a1.  b :- b1.

a :- a2.  b :- b2.

a :- a3.

(Other clauses are omitted)
```

In this program, first a process whose PTB is "go:-a,b" is generated and the goal a, the reducible goal in the goal sequence "a, b", is sent to the UU for reduction (the OR-fork count is 3). This results in three child processes being generated under the corresponding PCB. When one of these child processes returns the solution "a:-a1", a new PCB-PTB pair (go:-b) is generated and the goal b in turn becomes reducible. Figure A-2-1 illustrates the relationship between processes after the goal b is sent to the UU and two processes in OR-fork relation are generated.

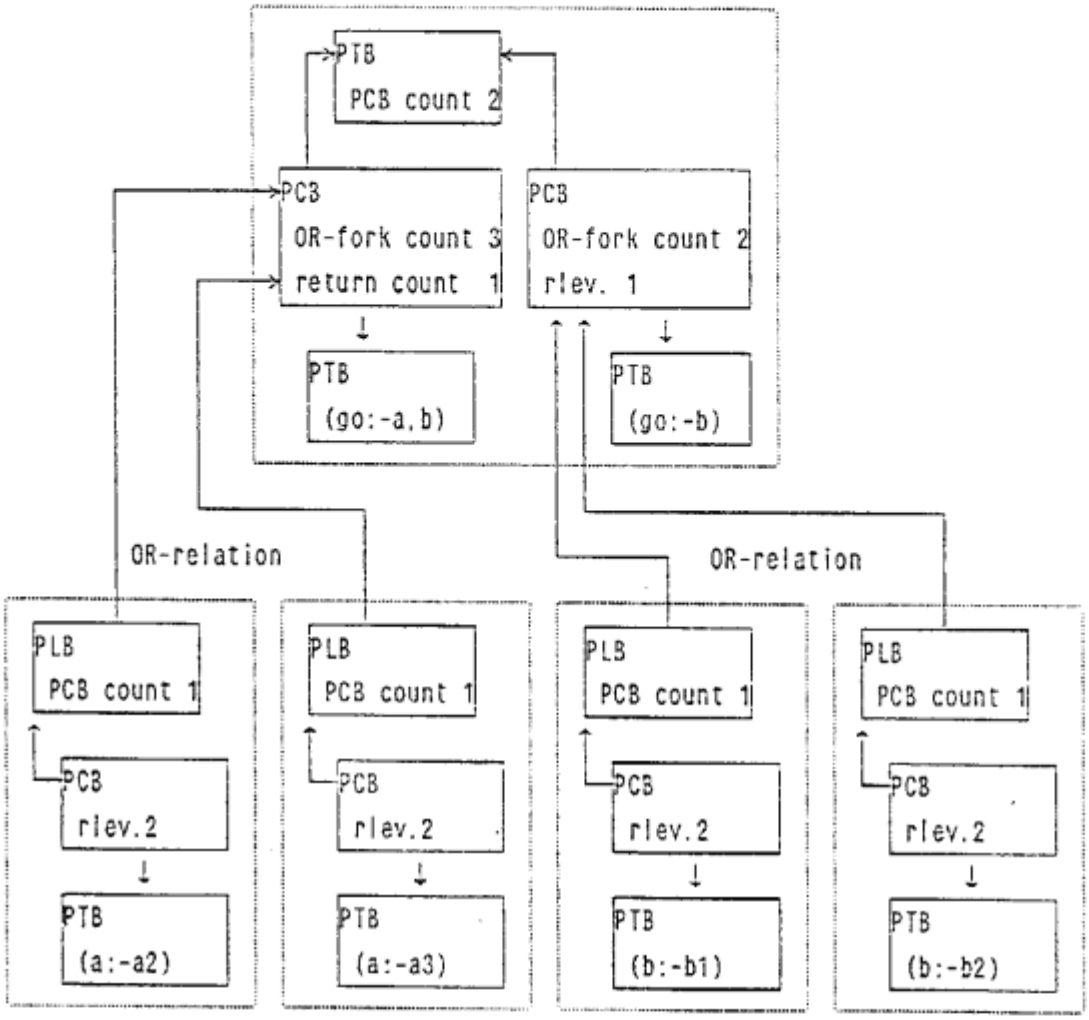


Fig. A-2-1

(Unless otherwise specified, the return count is 0 and the OR-fork count is undefined. "rlev." is an abbreviation for the reduction level. A dotted square indicates a process ; a process is stored in the Process Pool in the same IM.)

(Example A-2-2) For Concurrent Prolog

programs, as follows:

```
?- go.

go :- true | a, b.  (",," is a parallel AND-operator.)

a :- ag1 | ab1.  b :- bg1 | bb1.

a :- ag2 | ab2.  b :- bg2 | bb2.

(Other clauses are omitted)
```

In this program first "go" is executed. When "true" is successfully processed, the commit operation is performed and the process is updated. Then goals a and b are AND-forked to become different processes, which are then executed in parallel. Figure A-2-2 illustrates the inter-relationship between processes immediately after reduction is performed on a and b. In Concurrent Prolog, OR-related clauses become PCB-PTB pairs under the same PLB (i.e., they are stored in the same IM) and complete for execution of their guard parts.

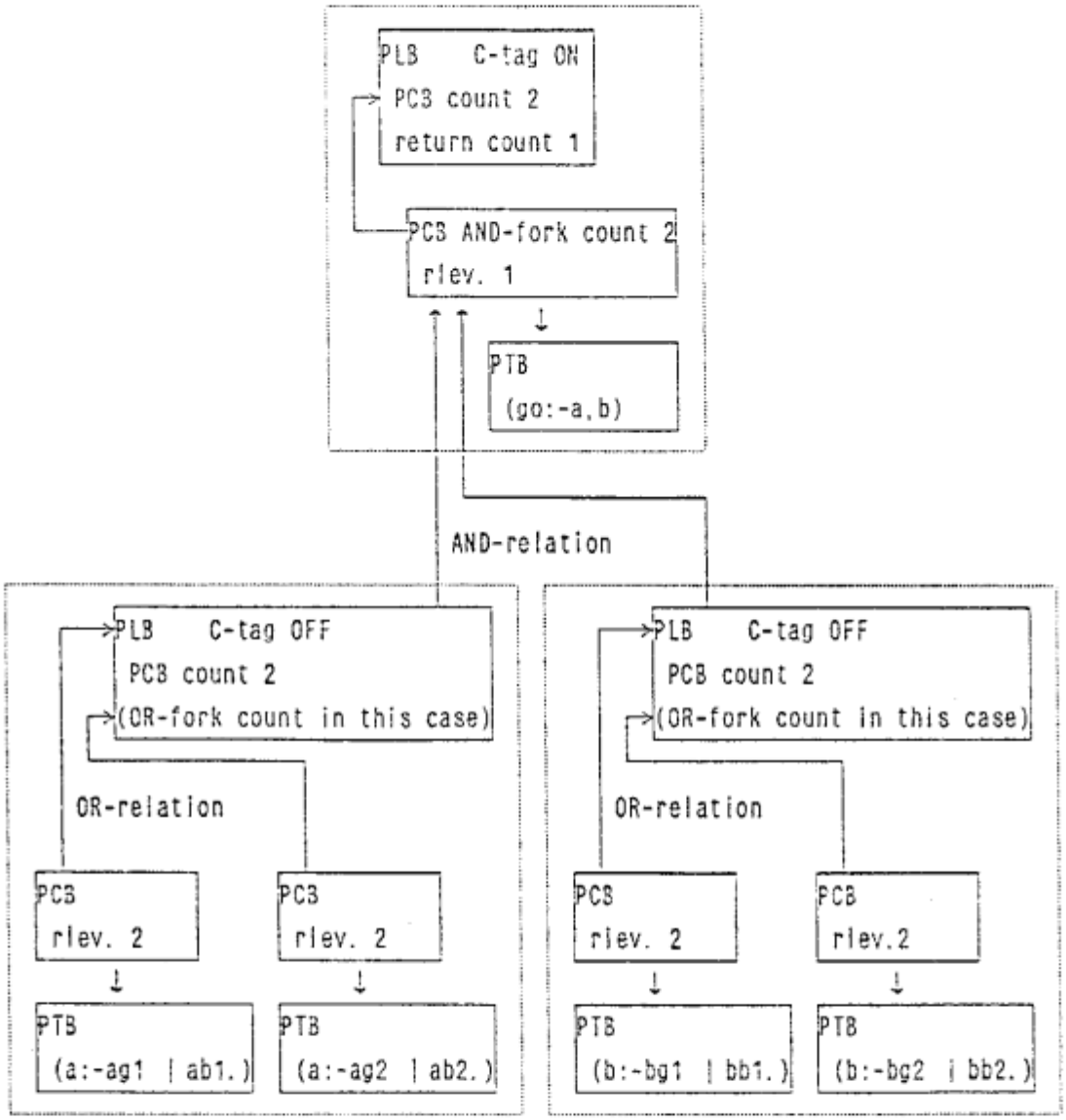
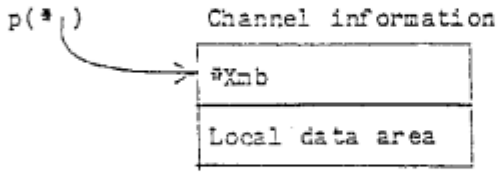

PLB C-tag ON
PC3 count 2
return count 1
PCB AND-fork count 2
rlev. 1
PTB
(go:-a,b)
AND-relation
PLB C-tag OFF
PCB count 2
(OR-fork count in this case)
OR-relation
PCB
rlev. 2
PTB
(a:-ag1 | ab1.)
PCB
rlev. 2
PTB
(a:-ag2 | ab2.)
PLB C-tag OFF
PCB count 2
(OR-fork count in this case)
OR-relation
PCB
rlev. 2
PTB
(b:-bg1 | bb1.)
PCB
rlev.2
PTB
(b:-bg2 | bb2.)

Fig. A-2-2

[Appendix 3]

Assume that a goal sequence p(X), c(X?). In this case, X is a channel. Then the PTB of p(X) can be described in simplified form as follows:



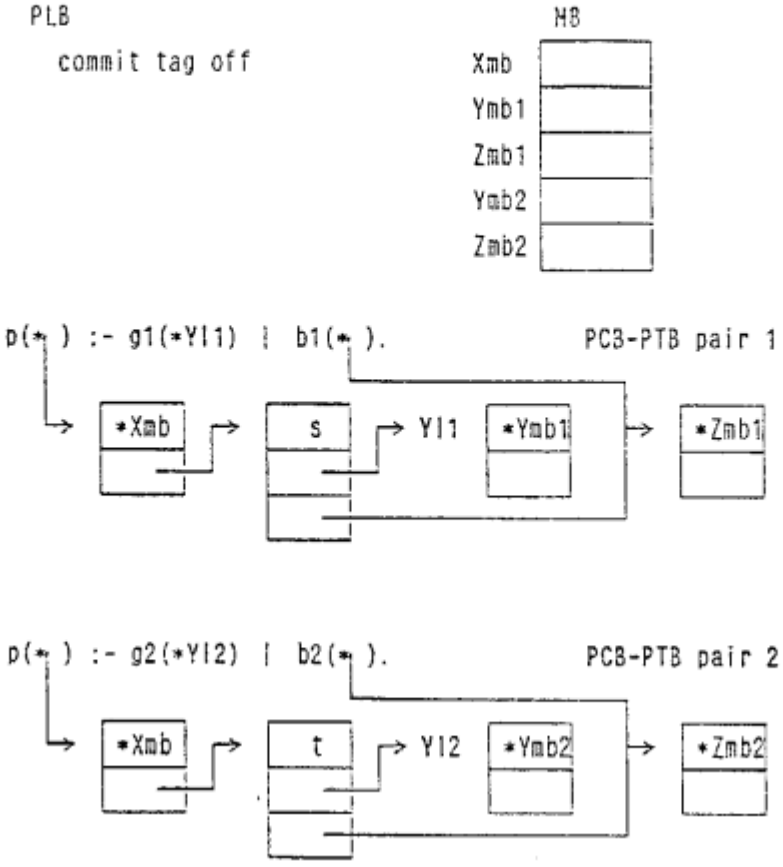
(Hereafter abbreviated to p(**Xmb).)

Clauses in OR-relation

```
p(s(Y,Z)) :- g1(Y) | b1(Z).
p(t(Y,Z)) :- g2(Y) | b2(Z).     (Y and Z are not yet channels)
```

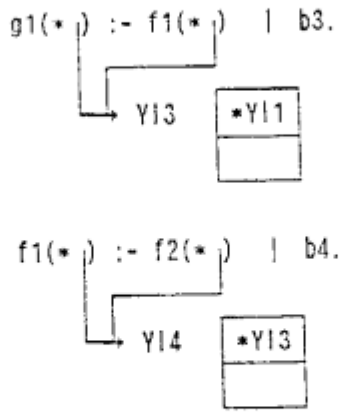
Since the characteristic of a channel is inherited at unification with goal p(**Xmb), Y and Z become channels and a cell is allocated on the MB for each OR-clause. By contrast, Y in guard goal g1 and g2 is not used as a channel at a child process level, because it has no guard goal in a concurrent relation. Therefore, Y in g1 and g2 has no new cell acquired on the MB and g1 and g2 share the channel information of Y with the head side. (In PIM-R, arguments are copied eagerly at reduction.) The unification results are shown below.

When g1 and g2 succeed in unification, these PCB-PTB pairs check the C-tag in PLB. Only one PCB-PTB pair which first turns on the C-tag can perform the commit operation and write local data in Xmb on the MB. If the guard goal g1 is nested as in the following example, f1 and f2 share the channel information with the head side, since they have no guard goal in a concurrent

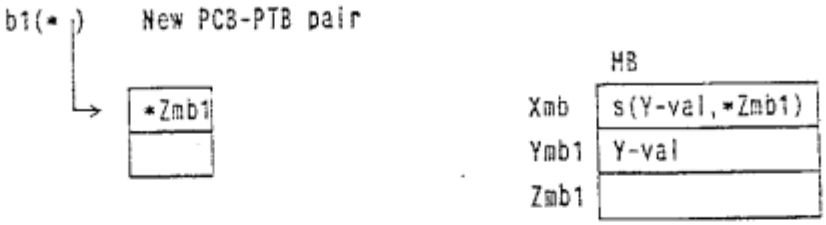


relation. When a PCB-PTB pair succeeds in unification, the contents of each local data area are returned as a binding environment to the local data area of the parent PCB-PTB pair.

When g1(*Y1) succeeds in unification and the commit operation is performed, the local data is written in Xmb and Ymb on the MB. When a suspension happens for a nested guard and a PCB(SPCB)-PTB pair is created, the channel information is traced backward to access the channel cell on the MB. If a value has already been written in the cell, it is activated; if not, the SPCB is recorded on the Suspend Process List.



Assume in this example that the guard of PCB-PTB pair 1 first succeeds in unification and the local data value of Yi1 at that time is Y-val. After the commit operation, the MB is updated as follows:



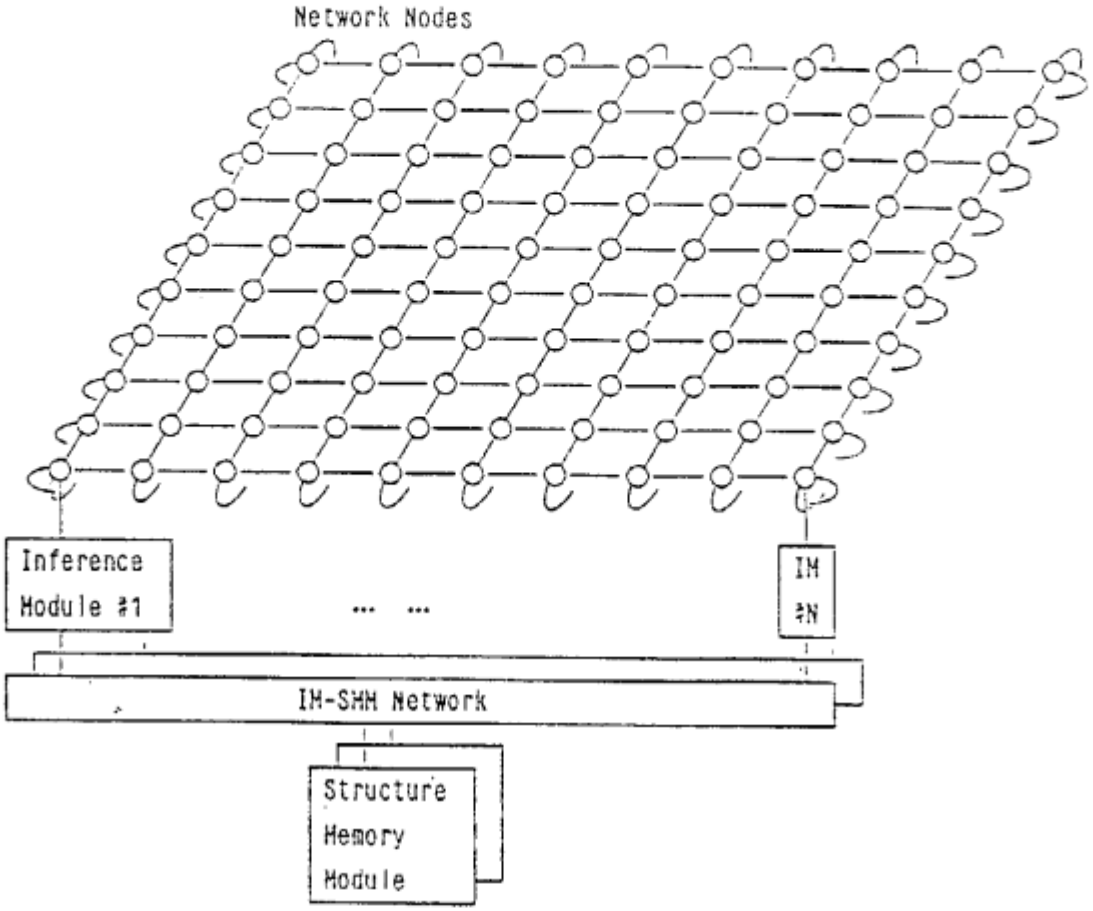


Fig. 3.1-1 Conceptual configuration of PIM-R

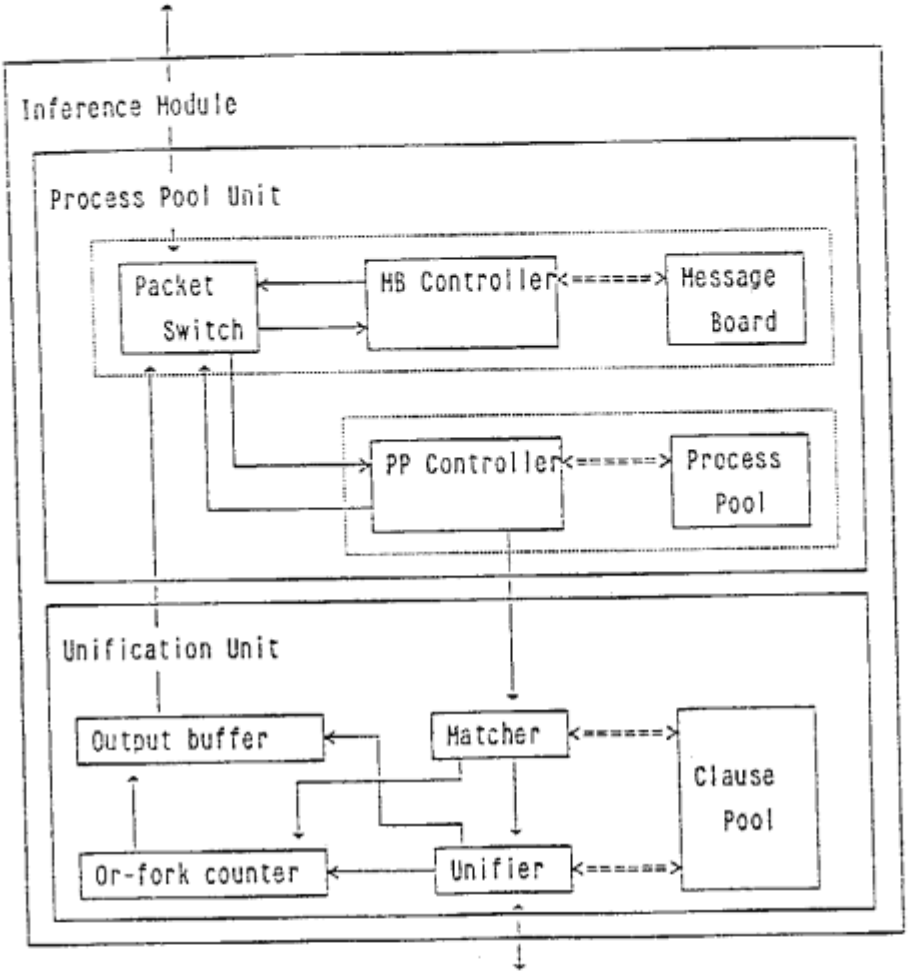


Fig. 3.1-2 Inference Module configuration

	Int	Clause length	Header
#0	Int	Head address of structure area	
	Int	Head address of literal header	
	Int	Number of variables	Variable area
		:	
	Int	Literal count	Literal header
	Type	Head literal	
	Type	Body literal N	
		:	
	Type	Body literal 1	
		:	Literal area
		:	Structure area

Fig. 3.1-3 The configuration of the Clause Definition Block
(Type is either Poi, Lit, Sym, or Para)

Int	23				Clause length	
Int	21		#0		Head address of structure area	Header
Int	5		#1		Head address of literal header	
Int	2		#2		Number of variables	
Var1	X		#3			Variable area
Var1	Y		#4			
Int	3		#5	#0	Literal count	
Lit	4	p	#6	#1	Head literal	Literal header
Lit	8	r	#7	#2	Body literal 2	
Lit	12	q	#8	#3	Body literal 1	
Int	3		#9	#4		
Poi	p		#10	#5	Head literal	
Int	1		#11	#6		
List	0	[X | Y]	#12	#7		
Int	3		#13	#8		Literal area
Poi	r		#14	#9	Body literal 2	
VarR	3	X	#15	#10		
VarR	4	Y	#16	#11		
Int	3		#17	#12		
Poi	q		#18	#13	Body literal 1	
Int	1		#19	#14		
VarR	3	X	#20	#15		
VarR	3		#21	#0	CAR Element	Structure area
VarR	4		#22	#1	CDR Element	

Fig. 3.1-4 Clause Definition Block of p(1,[X | Y]):- q(1,X) & r(X,Y).

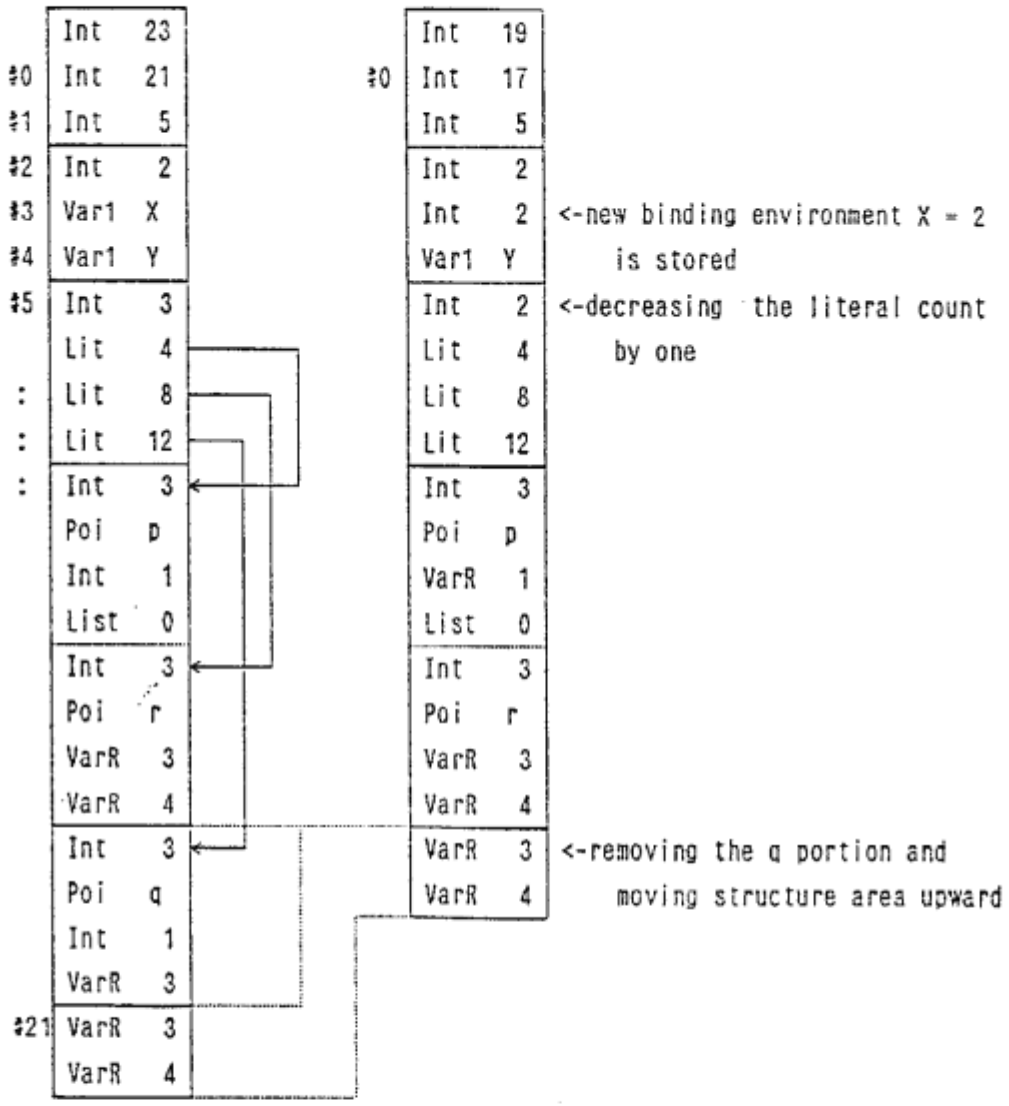


Fig. 3.1-5 Reverse compaction processing

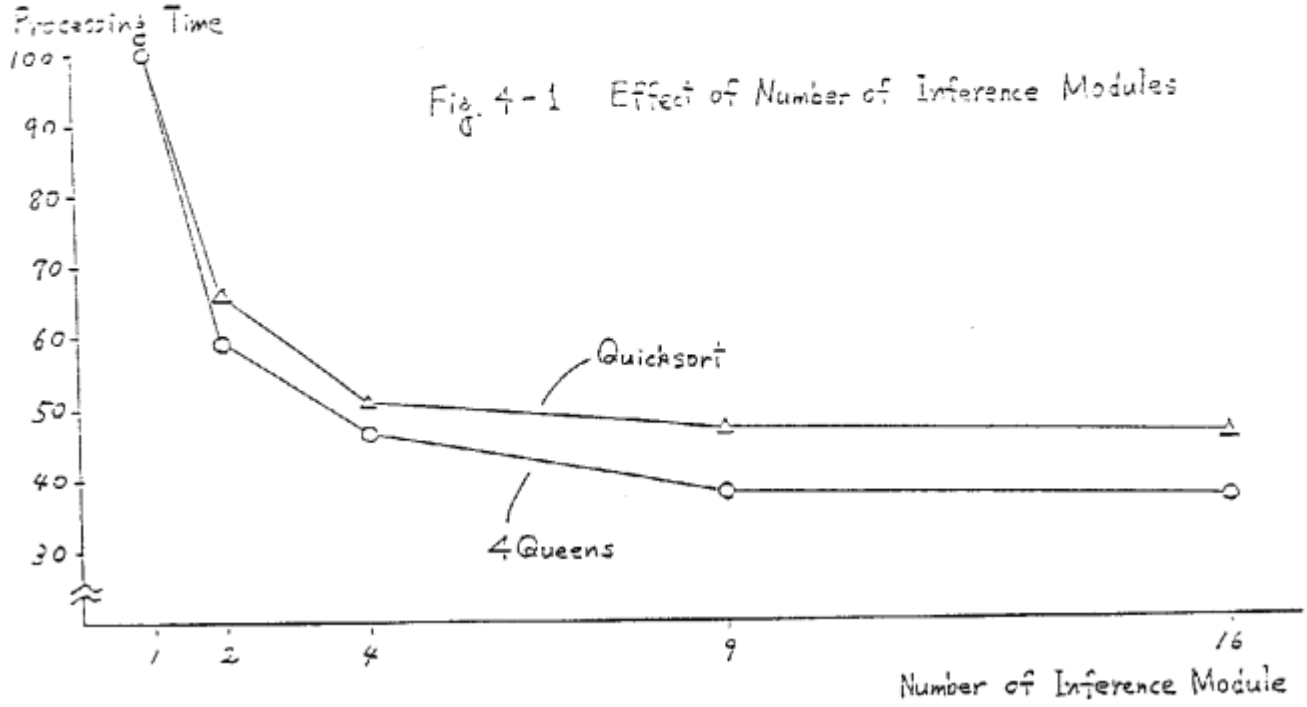


Fig. 4-1 Effect of Number of Inference Modules