PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

THE ARCHITECTURE OF A PARALLEL INFERENCE ENGINE – PIE –

Tohru MOTO-OKA, Hidehiko TANAKA
Hitoshi AIDA, Keiji HIRATA, and Tsutomu MARUYAMA

Department of Electrical Engineering, The University of Tokyo
Bunkyoku, Tokyo 113, JAPAN

ABSTRACT

This paper presents a highly parallel computer architecture which is oriented toward logic programs. One of the execution models of logic programs is the goal rewriting model based on OR parallelism. In this model, goals are independent of each other and stored in a goal pool. Each unify processor fetchs a goal, unifies it with definition clauses, and generates a few new goals. The Parallel Inference Engine (PIE) which implements this model has a highly modular architecture that interconnects 100s ~ 1000s inference units with two level networks. Several kinds of control mechanisms are introduced for the consistent and efficient execution of logic programs. This includes the unification policy control regarding goal selection and literal selection, the global control mechanism to implement guard and NOT operations, the distributed load leveling mechanism among unify processors, the process optimization scheme to reduce the goal size, and the sharing mechanism for structure data. The kernel part of unify processor is designed and implemented through TTL ICs. From this implementation, it is found that the unification operation can be done in 5 ~ 7 machine steps/cell. Software simulators are used for the evaluation of control mechanisms and of total system performance. Several control schemes are suggested by the results. Using a few test programs in PROLOG, the total system performance of PIE with 256 units is shown to be 170 times higher than the single processor machine. Lastly, extensions of PIE basic model including the elimination of duplicated goals and the set operations are discussed.

1. INTRODUCTION

The Fifth Generation Computer Systems are considered to be knowledge information processing systems, which are based on logic programming. The architecture of the fifth generation computers should support the efficient execution of logic programs. As logic programs are thought to have a lot of parallelism in execution level, we would be able to expect substantial speed-up through the parallel execution of logic programs. Up to this time, there proposed a few execution models for parallel inference processing such as AND-OR process model (Conery and Kiblar 1981), reduction model (Darlington and Reeve 1981), goal rewriting model (Goto et al. 1982, 1984), data flow model (Amamiya and Hasegawa 1983, Ito et al. 1983) and so on (Ciepielewski and Haridi 1983). However, concrete architecture proposals are very few, and evaluation results of the architectures are not reported at all. In this paper, we propose and evaluate a highly parallel computer architecture which is oriented towards logic programs. The execution model of logic programs is based on the goal rewriting model which we proposed before (Goto et al. 1982). As this model uses the OR parallelism, goals are logically independent of each other. However, if goals share some data physically, it makes the parallel execution difficult. So, the major part of each goal is created (copied) physically independently also, in this model. The machine which implements this model can be seen as a direct execution machine of logic programming languages such as PROLOG. We call this machine, Parallel Inference Engine – **PIE**. Section 2 summarizes the model and describes the basic architecture of **PIE (PIE-I)**. Section 3 is the description of control algorithms used to enable the consistent execution of parallel inference and to improve the execution efficiency. Section 4 shows the hardware implementation results of a unify processor which is the kernel part of **PIE**. Section 5 describes the software simulator of the basic architecture, and shows the evaluation results with a few control algorithms as parameters. Section 6 introduces the structure memory concept to improve the overhead of **PIE-I**, and makes clear the architectural requirements to incorporate the concept. Section 7 proposes the second version of **PIE (PIE-II)** based on the evaluation results of **PIE-I**, and

discusses the further study points. Section 8 is the conclusion of this paper.

2. THE BASIC MODEL OF PARALLEL INFERENCE ENGINE – PIE

2.1 Parallel Inference Model

From the analysis of inference operations, we can distinguish 4 kinds of parallel execution as follows:

(1) unification with multi clauses of the same predicate.
(2) multi new goals to be processed.
(3) multi AND literals which constitute a goal.
(4) multi arguments in a literal unification.

(1) is the parallelism found when several OR goals are created from a goal. On the other hand, (2) is the parallelism due to the fact that we have simultaneously many OR goals which should be solved. (3) is the AND parallelism. (4) is the inter-argument parallelism. From the computer architecture point of view, (1) can be implemented comparatively easily through providing several literal-unifiers. (2) can be implemented by having many unify processors which handle the unification of goal level. Though (3) can be implemented easily when the literals have no shared variables, it is difficult to implement as it needs consistency check generally. However, we can get other kind of parallelism through pipeline control, when passing the result of a unification of a literal to the unification operation for the other literals. (4) can also be implemented by having several argument-unifiers in each literal-unifier, though shared variables make the parallel operation complicated just like the (3). As the first step toward the parallel inference machine, we defined the basic model of **PIE** as follows:

(1) Sequential unification with multi clauses of the same predicate;
(2) Many unify processors are available;
(3) AND operations are processed sequentially;
(4) Unification of each argument is processed sequentially;
(5) New goals are independently created of each other (copy).

As the extension to the multi literal-unifiers will be straightforward, we assumed to have a single literal-unifier for each unify processor as the basic model. Items (3) and (4) result from the fact that consistency check is too heavy operation to justify the parallel execution, and that the speed-up gain of inter-arguments parallelism would be 2 or 3 at most. Item (5) is a tentative assumption to make the first machine model simple, which will be removed in section 7.

2.2 Machine Organization of PIE-I

Based on the model described previously, we designed the machine organization of **PIE** as Fig.1 (**PIE-I**). Definition Memory (DM) stores all definition clauses (whole programs). Memory Module (MM) stores the goal representations (called goal frames). Unify Processor (UP) fetches a goal from MM and several candidate clauses from DM, unifies them, generates new goals, and returns them to MMs. Activity Controller (AC) creates and maintains a part of the inference tree that is local to the AC, and controls the activity with communicating each other. Each Inference Unit (IU) is composed of a DM, a UP, a MM and an AC. Activity Manager (AM) is a

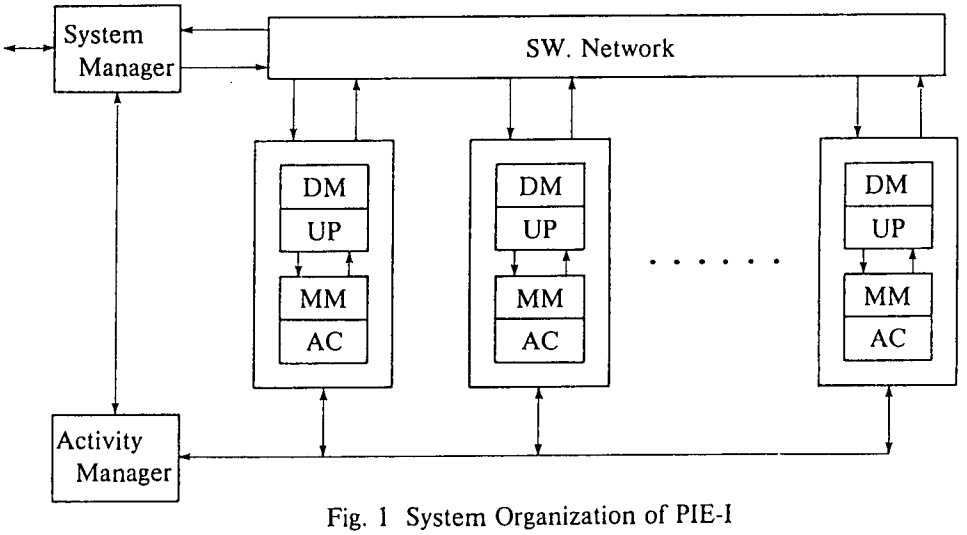

Fig. 1 System Organization of PIE-I

central controller for all ACs. System Manager is a central controller for the whole **PIE** machine, which controls the input-output as well. This organization of **PIE-I**, is a straightforward implementation of the basic model, and serves as a foundation to build up a further refined version.

2.3 Activity Control

The goal reduction process of machine **PIE** is to fetch a goal frame from the goal pool, to unify it with all candidate clauses of the same predicate for a literal of the goal, to create several new goals, and to return them into the goal pool. This cycle makes up the substantial part of inference process. Within this cycle, several control items can be distinguished, which are discussed in next section.

3. CONTROL ALGORITHM OF PIE-I

3.1 Unification Policy

Since all goals in the goal pool are basically independent, we have a freedom to select goals to be processed when the number of goals in the goal pool exceeds the number of inference units. We have studied four strategies for goal selection as follows:

(1) FIFO,
(2) depth-first,
(3) breadth-first,
(4) conclusion precedence.

(4) is a strategy to give precedence to such goals that seem to be nearer to the conclusion (true or false). That is, when we assign to each literal a level number which designates the depth in the inference tree, the highest priority is given to such a goal that the minimum level number of its goal literals is the maximum among all goals. In general, depth-first selection finds a solution earlier, while FIFO or breadth-first completes total search faster. On memory usage, depth-first requires less amount of memory to store waiting goals. The selection strategy (4) seems better at both time and space requirements than others, but it requires a little more complicated selection mechanism.

We have also a freedom in selecting the literal to be resolved in a goal frame. In most application programs, it is natural to select left-most literal, and it is good enough compared with the cost of literal selection in some complicated method. However, in some application programs, such as generator-consumer type programs, some literal selection mechanism should be equipped with. We have examined a simple literal selection method based on an annotation concept, which is similar to read-only annotation in Concurrent Prolog, and found it works well at a low cost (Aida et al. 1984).

3.2 Load Leveling

It is always a quite difficult problem how to use as many processors as possible efficiently in multi processor systems. In **PIE**, the leveling of processing load is done by sending newly generated goal frames to other units. Since sending goal frames costs network delay, it is a trade off of load leveling and network cost. We have examined three strategies for load leveling. One is so-called 'first-self', in which method the first one of newly generated goal frames is stored into its own MM, while other goal frames are sent out to other MMs. The second one is so-called 'empty-self', in which method newly generated goal frames are stored into its own MM if and only if the number of goal frames in its MM is less than a threshold which is specified by the activity manager. The third one is a random scheme which distributes newly generated goals to MMs randomly.

3.3 Global Control

To implement unification policies as described in 3.1, there should be some mechanism to associate goal frames with each other. The derivational relation among goal frames is especially important when we implement extended control features of logic programming languages, such as negation-as-failure or guarded clauses. The simplest way to implement such mechanisms is to record the derivational relation in a tree form. In **PIE**, this tree is called inference tree and stored in dedicated memories in ACs. ACs manage the tree by sending 'node control commands' one another.

The inference tree is expanded by the information included in the header part of a newly generated goal frame. The header includes parent node id. in the inference tree and the attribute of the newly generated node. The MM Controller picks up these information from the goal frame and passes them to the AC. Then the AC sends a 'son' command to the AC which manages the parent node, and establishes a parent-son relation between the parent goal frame and the newly generated goal frame. UP makes a dummy goal frame which includes a control information when no more clauses can be unified with the input goal frame. The control information includes the number of newly generated goal frames, and the new attribute for the parent node. When AC receives these information, AC changes the state of parent node and sends a command to the MM

to delete the goal frame itself in order to save memory space. Therefore, only goal frames waiting for reduction and just under reduction are stored in the goal pool.

When the resolution of a goal frame ended in success (null clause) or failure, the corresponding branch of inference tree is trimmed from the leaf toward the root. If activity control is not busy, unnecessary relay nodes each of which have only one son node are also removed by exchanging four commands between ACs.

3.4 Process Optimization

In conventional inference operation, goal size becomes larger with the inference steps. However, at the reduction stage in UPs, all the information unnecessary for later reduction (such as intermediate variables) can be discarded. Therefore, the size of each goal frame can be expected to preserve its original size by using this size reduction mechanism, while the number of goal frames in the goal pool is dramatically changed.

UP may not return its last producing goal frame to MM, but may begin the next unification cycle with it immediately. Especially, if the literal to be resolved is deterministic (such as arithmetic evaluable predicates), there is no transfer of goal frame between UP and MM, and the network delay can be saved. The deterministic predicates amount to about half of the total reduction, in the <8-queens> program execution, for example. Accordingly, we can expect saving the round trip delay time by this 'short-cut' scheme.

4. UNIFY PROCESSOR

4.1 Design of Unify Processor

To get time values of a few elementary operations which will be used for the system evaluation through simulation, we built an experimental UP by TTL ICs. A UP performs unification of literal pairs of the Goal Frame (GF) and the Definition Template (DT, definition clause stored in DM). The unification algorithm used in UP is currently almost the same as that of DEC-10 PROLOG. If the unification succeeds, UP gathers only the necessary information from the GF and the DT, and then generates a new GF. This operation is called size reduction. The main features of size reduction are as follows.

(1) Elimination of unreferred cells
(2) Elimination of bound variables
(3) Renumbering of variable numbers

Fig.2 is the block diagram of a UP. The function of each block is as follows.

(1) Local Memories

A GF is copied from the Input Buffer to the Goal Frame Local Memory (GFLM). A DT is transferred from the DM to the Definition Template Local Memory (DTLM).

(2) Unifier

The unifier unifies the selected goal literal of the GF in GFLM and the head literal of the DT in DTLM. After the unification ends in success, variable substitutions remain in the Local Memories.

(3) Reducer

Size reduction is performed by the Reducer. It produces a new GF into the Output Buffer from the GF and the DT using the variable substitution information.

(4) System Predicate Processor

The System Predicate Processor executes built-in system predicates, such as 'add'.

(5) Input / Output Buffers

Input and Output Buffers are buffers for the interface with the networks. These buffers enables the pipeline operation of the processing in UP and the transfer of GFs in the network.

(6) Interfaces

Interfaces are with MM → UP network, UP → MM network, the DM, and the system manager.

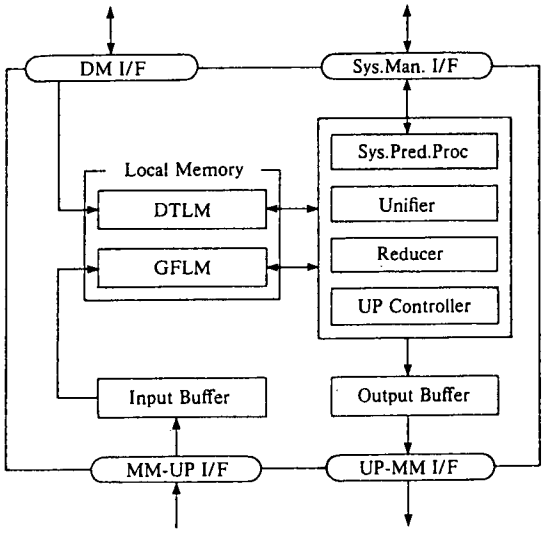

Fig. 2 Block Diagram of Unify Processor

4.2 Hardware Implementation.

We implemented a UP for experimental purposes. The UP Pilot Machine is divided into UNIRED and SVP. UNIRED is a special hardware for unification and reduction and corresponds to the Unifier / Reducer, Local Memories, and Input Buffers in Fig.2. SVP is a general purpose microprocessor system (68000 CPU) and simulates the functions of the System Predicate Processor and Interfaces.

Special features of UNIRED are as follows:

(1) tagged architecture,
(2) horizontal microprogrammed control (58 bit / micro instruction),
(3) dedicated internal busses for GF and DT,
(4) dereference of variables by the hardware,
(5) eleven functional registers, six of which are counters,
(6) multi-way jump facility according to the contents of the registers,
(7) three hardware stacks which are pushed/popped at the same time,
(8) high speed local memories which can be accessed simultaneously.

UNIRED is made of about 500 TTL ICs and 128 CMOS RAMs (access time 45 ns). Table 1 and 2 show the implementation results of UNIRED at the microprogram level. A success unification requires 23 μ-steps (approximately 5μs) for <6-queens> and 74 μ-steps (15μs) for <PENT>. A failure of unification is detected after 10 ~ 18 μ-steps (3μs). A reduction requires 185 μ-steps (37μs) for <6-queens> and 1565 μ-steps (313μs) for <PENT>. It needs 4.6 ~ 6.3 μ-steps (0.9~1.4μs) to reduce one cell of a GF.

Table 1. The Unification Clocks in UNIRED

program	average μ-steps	
	success	failure
6-queens	23.4	10.6
equiv2	26.2	10.1
PENT	74.0	9.9

Table 2. The Reduction Steps in UNIRED

program	average GF length	average reduction μ-steps	reduction μ-steps per cell
6-queens	35	185	5.25
equiv2	57	267	4.61
PENT	248	1565	6.29

5. SIMULATION OF PIE-I

5.1 Simulation Model and Simulator

Software simulators were made based on the machine organization of **PIE** described in 2.2. The operation of each UP is divided into 3 phases, i.e., preparation phase, candidate clause selection phase, and unification and reduction phase. From the implementation results of UP, it is found that the total duration of preparation and selection phases are nearly constant, and the duration of the unification and size reduction phase is proportional to the GF size. As some multistage network such as omega network is assumed for the distribution network, the transmission delay is proportional to the logarithm of the number of IUs and to the GF size. The switch control time is negligible compared with the transmission time of a GF, as the GF size is pretty large (about 100 cells). The delays of other networks can be set as constant, as their effect on the overall performance is rather small.

The simulation parameters for **PIE-I** are assumed as follows:

(1) Each module in IU has unbounded buffers in its each input port;
(2) The 3 phases of UP take 100 machine clocks, 50 clocks and 7 clocks/cell, for their processing, respectively, and every μ-step takes one machine clock;
(3) The delay time of distribution network is 4 × ($\log_4$n) × (goal size in cell) clocks, where n = number of the IUs, and cell = 4 bytes; The conflict of GF transmission on the network is not taken into account;
(4) GF selection strategy is FIFO;
(5) Load leveling strategy is 'empty-self';
(6) The processing time for the global control described in 3.3 isn't considered.

This simulator is written in language C and partly in assembler of VAX, and has about 9,500 lines. In order to simulate concurrent processes, several process control functions are provided in this simulator. This simulator can simulate 256 IUs.

5.2 Simulation Results

(1) Total performance

Fig.3 shows the relative speed against a single UP. The total performance of **PIE** of 256 IUs for the test program <8-queens> is about 170 times faster than the single processor machine. The performance for the test programs <6-queens> and <LL2P> is saturated (<LL2P> is a program of cryptarithmetic problem (LISP + LOGIC) × 2 =

PROLOG). From the measurement of maximum number of parallelism for each test programs, the main reason of the saturation for these test programs can be thought to be due to the shortage of the parallelism of the test programs against the number of UPs. This means that if the program have enough parallelism, **PIE** can exhibit high performance corresponding to the parallelism.

(2) Load leveling

Fig.4 shows the number of the working unifier/reducer in each simulation time. As shown in Fig.4, 'empty-self' is better than 'first-self'. In this simulation, the value of the threshold of 'empty-self' is always set to 1. The value of the threshold influences the traffic of the network and the total system performance. The method to decide the optimum value of the threshold should be examined through more precise simulations.

(3) Goal selection

Goal selection strategies described in 3.1 are examined. Fig.5 shows the total number of the GFs in MMs and the time (designated by *) when the answer was found. On memory usage, the strategy (4) seems better in this test program. In practice, goal selection strategy should be changed dynamically according to the memory usage.

(4) Global control

Through the simulations of the global control mechanism, it is found that when one GF is created, 4 ~ 6 node control commands are generated at the busiest time. This means that ACs must execute these commands in 100 ~ 200 clocks. It is not difficult to realize this execution speed on real machine.

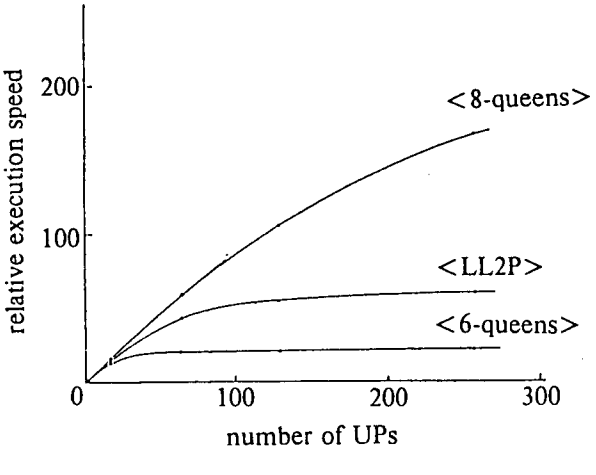

Fig. 3 Relative Execution Speed of PIE-I

The traffic of the node control commands depends on the load leveling strategy. 'Empty-self' tends to increase the command traffic compared with 'first-self'. However, the number of commands transferred through the command network is at most 8 per one AC in each 2000 clocks, because the elimination of unnecessary relay node keeps the size of the inference tree small. The command size is no longer than 4 cells. Accordingly, if 10s IUs are available, the command network may be implemented by bus.

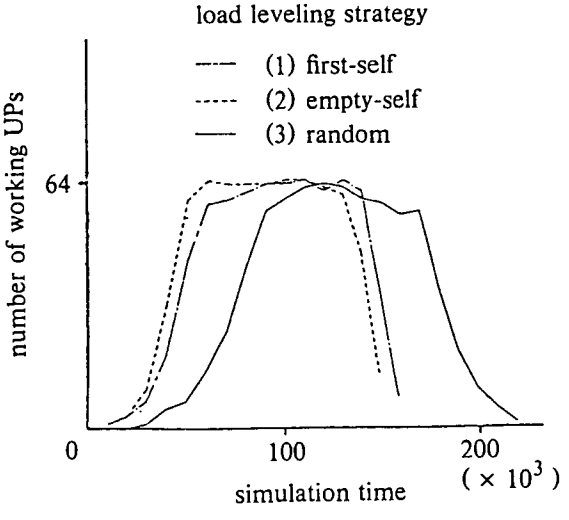

Fig. 4 Effect of Load Leveling
<LL2P> UP = 64

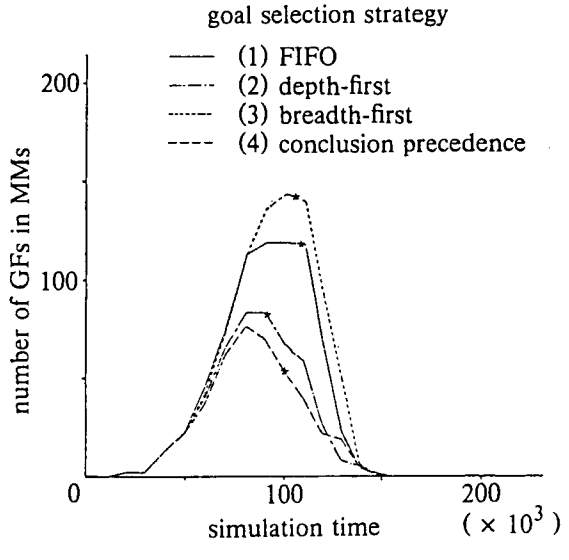

Fig. 5 Effect of Goal Selection
<LL2P> UP = 64

Since the traffic for global control is little as stated above, the total system performance will not be much influenced even if the processing load of global control is taken into account.

(5) Size Reduction

Fig.6 shows the average GF size at each unification depth. The goal size can be preserved constant by incorporating the size reduction mechanism, while the size grows linearly with the unification steps when the size reduction is not applied.

6. ADDING THE DATA SHARING CONCEPT

6.1 Structure Memory Concept

From the previous section, it becomes clear that the GF size is a predominant parameter to decide the system performance, though the size can be kept not to explode through the size reduction mechanism. In this section, we introduce the structure memory concept to reduce the copying overhead. Though the conventional structure memory (Amamiya and Hasegawa 1983) stores all of the structure data, our structure memory stores only a part of the structure data so as to match the highly parallel architecture. Fig.7(a) shows the representation of a GF in **PIE-I** system, where the structure area includes all of the structure data as well as the undefined variables and ground instances.

As the first step toward data sharing, we divide the structure area into 2 parts: One is the part which includes undefined variables, and another is the part which does not. The first part is placed in the MMs the same as **PIE-I**, and exchanged between UPs and MMs in a lump. As the second part does not change the content any more, we placed this part in the structure memory which is shared by all UPs (Fig.7(b)). That is, the immutable structure data is placed in the shared structure memory, and accessed by UPs on demand basis. The size of immutable structure data which GF possesses changes dramatically depending on applications. But when unification is done, UP requires only a part of it, so the access to the structure memory occurs on a small amount basis. We decided this amount as a node data which constitutes structure data. We call this access on a node basis as 'lazy fetch'. When some new goals are generated, new structure data are created using a part of the structure data and some constants. New ground instances in the new structure data are stored in the structure memory. The rest of the new structure data is stored in MMs independently of each other, so that they are not shared. When the size of immutable part is large, we can expect 2 fold gains through the decrease of transfer data, and the decrease of size reduction time.

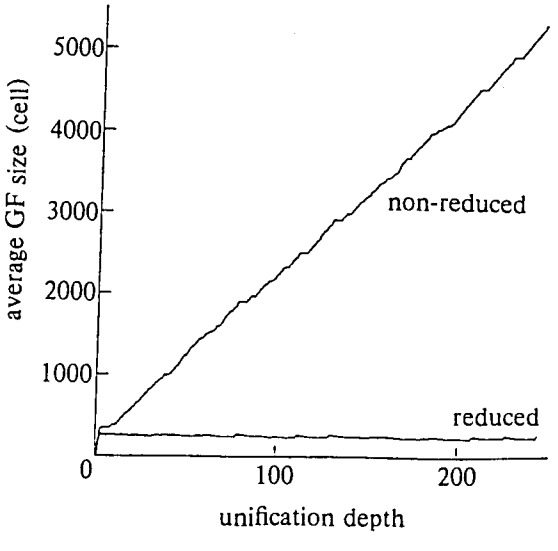

Fig. 6 Effect of Size Reduction
<PENT>

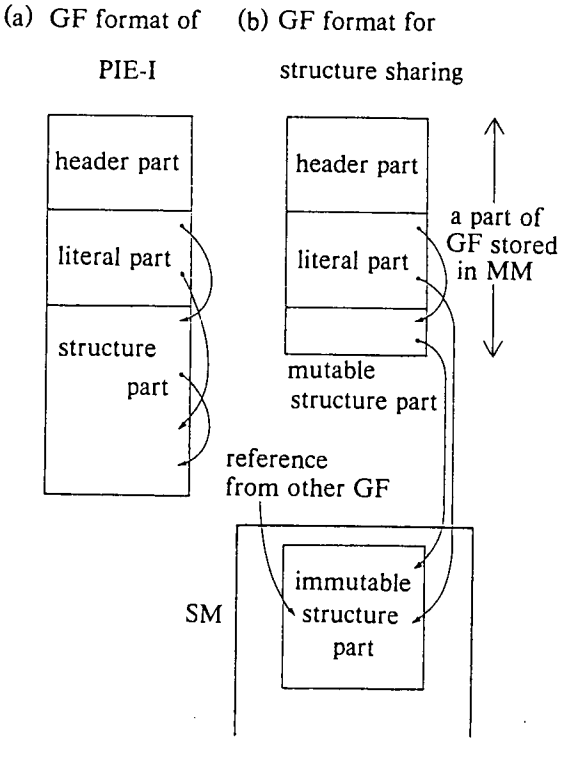

Fig. 7 Internal Representation of Goal Frame

6.2 Simulation

A simulation program is written to evaluate the effectiveness of structure memory mechanism. The program size is about 3.800 lines in language C. We measured the number of lazy fetch operations, the transferred data size, the size reduction time, and so on, for 3 test programs (<equiv2>, <PENT>, and <LL2P>).

As the results, the average size of transferred data between a MM and a UP decreased to 50%. The size reduction time decreased to a half to a tenth, depending on the applications. As this time is several times larger than unification time in **PIE-I** architecture, this decrease is expected to improve the system performance considerably.

Table 3 shows the number of lazy fetch operations. From these results, we can say that 70~96% of unifications don't need the structure memory access, and 1~2 Lazy Fetch (LF) operations per one unification are done when they are needed.

Table 3. Evaluation of Lazy Fetch Operations

program	number of unifications	unifications without LF (%)	average † LF operations
equiv2	7943	90.5	1.20
LL2P	19485	68.4	1.00
PENT	5865	95.6	1.93

† per one unification when needed

7. DESIGN OF THE SECOND MODEL

7.1 The Architecture of PIE-II

Based on the evaluation results of **PIE-I** and the structure memory, we designed the second model of **PIE (PIE-II)** as Fig.8 and 9. The modified points are

(1) 2 level modular architecture,
(2) introduction of structure memory,
and
(3) consideration of Input / Output path.

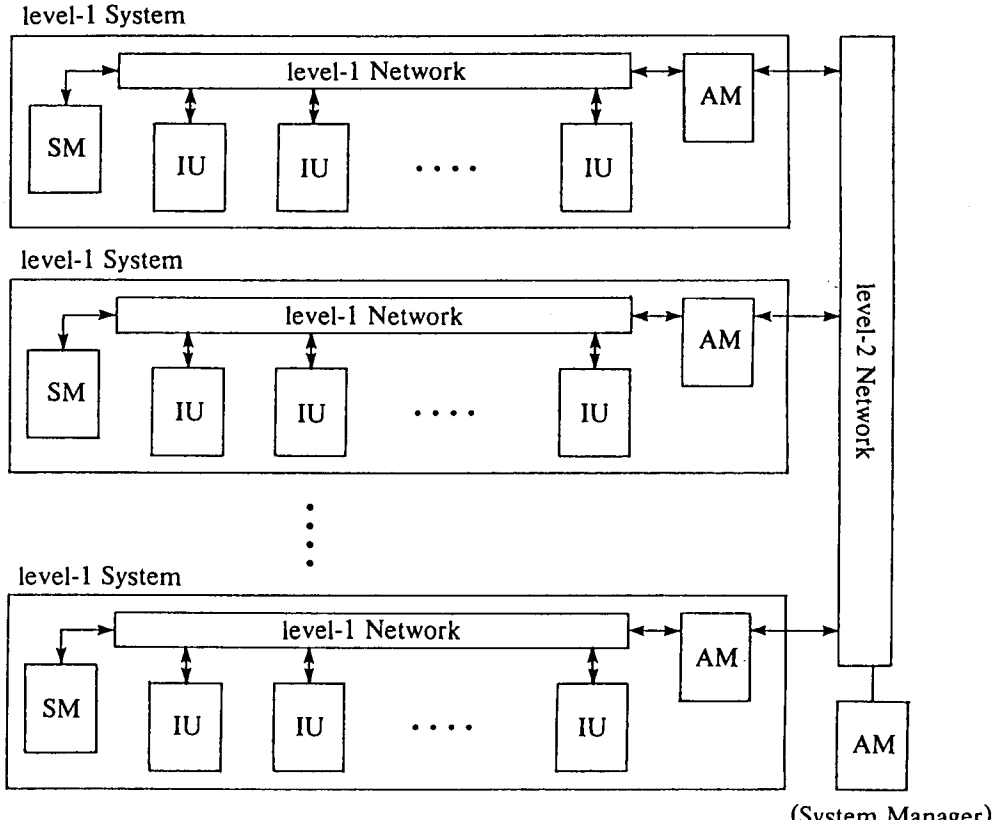

Fig. 8 The Global Architecture of PIE-II

Our target parallelism is more than 100. This value is too high to support a shared memory for the structure data. So, we divided the whole system into level-1 systems, each of which is made of Inference Units (IU) and a Structure Memory (SM). Each structure memory is shared by the IUs in the level-1 system. When some goals are distributed to other level-1 system according to the load balancing mechanism, the structure data referenced by the goals are added to the goal data, and transferred to the other level-1 system. As structure data is immutable in **PIE-II**, this copy operation can be implemented easily. Each IU is equipped with an AC. AM controls the ACs in the level-1 system to adjust the load balance among IUs and interfaces to the level-2 network. System manager is a central controller of the whole system.

To incorporate the structure memory, a Lazy Fetch Buffer (LFB) is provided in each IU. Each UP fetches structure data through the LFB which is a buffer to access the structure memory through Lazy Fetch Network (LFN). A few IUs in a level-1 system are Input Output Units (IOU), each of which includes an I/O Processor (IOP). The level-1 network of Fig.9 is made of 3 kinds of networks: LFN, Distribution Network (DN) and Command Network (CN). DN is used to transfer GFs from UPs to external MMs at processing stage so as to distribute load to other IUs and Definition Clauses between IOU and IUs at program loading stage. The CN is a network to exchange activity control commands among ACs, and between ACs and the AM, in the level-1 system.

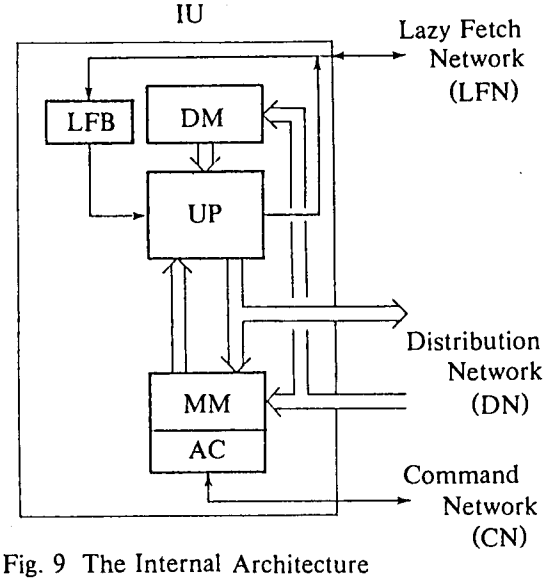

Fig. 9 The Internal Architecture of Inference Unit

The lazy fetch operations require quick responses from the structure memory, while the total traffic is about less than one tenth of GF traffic. Accordingly, low delay networks such as bus are suitable for LFN, and the number of IUs in a level-1 system should be less than 20. DN transfers a large size data (of the order of 100s bytes), while low delay characteristics are not required so strictly if the goal queue of each MM does not become empty. Some multi-stage networks such as omega network will be adequate for this purpose. As the traffic of activity control commands is low compared with the goal traffic, and does not make significant influence to the total performance, a conventional bus can be used for the CN. This low sensitivity to the performance comes from the loose coupling characteristics between unification processing and activity control. The level-2 network should support the 2 kinds of traffic for DN and CN. However, low delay requirement is not so severe that some high throughput network will be enough for it. We expect that the performance of 64 level-1 systems (hence 1024 IUs) will be a few hundreds times faster than the single IU system. As the operations of level-1 systems are independent of each other, the level-2 control is basically very simple.

7.2 Future Extensions

(1) Removal of unneeded goals

As the goals of **PIE** are processed independently, the goal pool may include duplicated goals. They can be removed except one of them. Furthermore, if the machine can memorize the results of each goal unification, we can use them to make the inference process effective by removing such goals that are equivalent to the failed goals.

In some application programs, the same GF appears more than once in the execution of a input goal. If the capacity of the goal pool is large enough, we could leave GFs even after their reductions complete, and check whether a newly generated GF is already processed before. If the GF is found to be redundant, it is unnecessary and can be removed in most cases. If the GF is under a negation-as-failure node, however, the nodes of inference tree should be merged into one rather than simply removed.

Redundancy check may be done in a literal level instead of a GF level. Memorizing failed goal literals requires less memory and more practical, while memorizing successful goal literals is a kind of knowledge acquisition.

(2) Set operation

In some application programs, it is desirable to collect all the solutions in a set. If the search of all the solutions should be done in parallel, some kinds of AND parallel processing mechanism should be introduced to make up the solution set.

Suppose a following example:

```
?- setof(X,p(X),S),q(S).
p(X) :- a(X).
p(X) :- b(X).
```

The GF including 'setof' literal can be divided into child 'setof' GFs corresponding to each definition of p(X), and the rest parts with 'setmerge' literal in its head, in UP. Since child 'setof' goals and the rest parts are AND related with shared variables, the rest parts are suspended by AC and wait the finish of the execution of child 'setof' goals. When a subset of solutions is gained by one of the child 'setof' goals, the AC picks up the pointer to the set (stored in the structure memory) from the final GF and stores it in the corresponding argument cell of 'setmerge' literal. When the execution of all sub 'setof' goals is finished, AC activates the rest parts of the goal.

If the parallel execution of the rest parts with 'setof' goals is required, it should be supported by higher level operation of SM such as 'cons' operation.

8. CONCLUSION

In this paper, a highly parallel computer architecture for logic programs is proposed. This machine **PIE** is made of 100s ~ 1000s inference units. The inference operations which are done on unify processors in inference units are almost independent of each other. As the control mechanism is loosely coupled with the inference operations, very flexible control is possible in spite of the highly parallel architecture. The kernel part of the unify processor is implemented of TTL ICs. Software simulators are also made to evaluate the control mechanism and the total performance of the basic model of **PIE**. From the results, a few control schemes are recommended for each control items. The total system performance of **PIE** of 256 inference units is 170 times higher than the single processor. Some extensions of **PIE** are also discussed regarding the efficiency improvement through eliminating the duplicated goals, and the set operations.

Further study items are as follows: Incorporating more than one literal unifiers in a UP, the experimental productions of other units than UP, the system evaluation for larger **PIE** and for many other applications, and the interface clarification to the knowledge base machines.

ACKNOWLEDGEMENTS

The authors wish to thank Dr. Atsuhiro Goto (Now at Electrical Communication Laboratory), Messrs. Masanobu Yuhara (Now at Fujitsu), Shuichi Sakai, Naoki Hamanaka, Hanpei Koike, and Kenji Matsubara of our laboratory for their corporative works and helpful discussions.

REFERENCES

Aida, H., Goto, A., Tanaka, H., and Moto-oka, T.: On Utilization of Read-only Annotation in Goal-rewriting Model of Logic Programs, *Proc. of Biannual Meeting of IPSJ, 4H-5,* 1984 (in Japanese).

Amamiya, M. and Hasegawa, R.: A Logic Program Execution Mechanism Based on Data Flow Control, *Proc. Logic Programming Conference '83, 9.1, ICOT,* 1983 (in Japanese).

Ciepielewski, A. and Haridi, S.: A Formal Model for Or-Parallel Execution of Logic Programs, *Proc. IFIP 83,* pp.299-306, 1983.

Conery, J.S. and Kiblar, D.F.: Parallel Interpretation of Logic Programs, *Proc. Conf. on Functional Programming Languages and Computer Architecture,* pp.163-170, 1981.

Darlington, J. and Reeve M.J.: ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages, *ibid,* pp.65-76, 1981.

Goto, A., Aida, H., Tanaka, H., and Moto-oka, T.: The Basic Architecture of Highly-Parallel Processing System for Inference, *IECE, EC82-43,* 1982 (in Japanese).

Goto, A., Tanaka, H., and Moto-oka, T.: Highly Parallel Inference Engine PIE – Goal Rewriting Model and Machine Architecture –, *New Generation Computing, Vol. 2, No. 1,* pp.37-58, OHMSHA, 1984.

Ito, T., Onai, R., Masuda, Y., and Shimizu, H.: Data Flow Type Parallel Prolog Machine, *Proc. Logic Programming Conference '83, 9.3,* 1983 (in Japanese).