TR-160

The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D

by
Eiji Kuno, Noriyoshi Ito
(Oki Electric Industry)
Masatoshi Sato and Kazuaki Rokusawa
(ICOT)

March, 1986

THE ARCHITECTURE AND PRELIMINARY EVALUATION RESULTS OF THE EXPERIMENTAL PARALLEL INFERENCE MACHINE PIM-D

Noriyoshi Ito*, Masatoshi Sato
Institute for New Generation Computer Technology,
1-4-28 Mita, Minato-ku, Tokyo, Japan

Eiji Kuno, and Kazuaki Rokusawa**
Oki Electric Industry Co.,
4-10-12 Shibaura, Minato-ku, Tokyo, Japan

ABSTRACT

A parallel inference machine based on the dataflow model and the mechanisms to support two types of logic programming languages are presented. The machine is constructed from multiple processing elements and structure memories interconnected through a low-latency hierarchical network. The preliminary evaluation results of the experimental machine are also presented. The evaluation results show that the machine can exploit parallelism in programs.

1. INTRODUCTION

The authors are investigating the Parallel Inference Machine based on the Dataflow model (PIM-D) and the mechanisms to support the parallel logic programming languages. The main features of the PIM-D are listed below.

- dataflow-based architecture: The dataflow model assures the independence of operations being executed in parallel [7] [3] [9]. There are no side effects among the operations. Thus, the parallelism in the programs (OR-parallelism, AND-parallelism, and parallelism included in the unification operation) can be easily exploited by the machine.

- hierarchical network configuration: The machine is constructed from multiple processing elements and multiple structure memories interconnected through a hierarchical network. Communication among these units is performed via the hierarchical common busses. Each bus provides high-bandwidth, low-latency packet transmission facility; a packet can be transferred from one unit to another in one bus transmission cycle.

- support of two types of logic programming languages: The machine can support two types of logic programming languages, OR-parallel and AND-parallel Prolog, in a uniform manner; inter-process communication is performed by streams, that are represented by non-strict structured data in order to provide an asynchronous communication among processes.

There are several approaches to implement highly parallel inference machines. Conventional processors based on the von Neumann model, in which both program codes and data are stored in a memory unit, fetch and interpret program codes sequentially. If such processors are used as the processing elements of the distributed, parallel inference machine, context switching of processes or packet communication overhead caused by frequent remote accesses of shared structures or shared variables will significantly reduce system performance [2] [5]. The data flow machine assures independence of the operations being executed in parallel. The machine can easily exploit parallelism in such a distributed processing environment.

As the authors adopted the hierarchical network architecture, where all the processors are on the leaves of the n-ary tree hierarchy, the machine is easily expandable to any scale and can make use of the locality of the programs while exploiting parallelism. The main idea in implementing such process allocation is to vary the process distribution factor according to the computation load of the system as described in Section 4.

GHC (Guarded Horn Clauses) was chosen as the basic language of KL1 (Kernel Language 1): a parallel version of the kernel language of the fifth generation computers in ICOT [16]. It is an AND-parallel logic programming language implementing stream AND-parallelism. On the other hand, a class of all solution searching problems is easily described by OR-parallel logic programming language by using its nondeterminacy. The authors introduced the stream communication scheme for the OR-parallel language as well as the AND-parallel language [13]. Therefore, two types of logic programming languages and their interface can be supported on the machine in a uniform manner.

A detailed software simulator for PIM-D was developed to estimate the performance of the machine from a single-processor system to a large-scale system [14]. The results showed that

* N.Ito is currently with OKI Electric Industry Co., Ltd.
** K.Rokusawa is currently with ICOT.

the machine can exploit parallelism and that the performance is significantly increased when the number of the processing elements is increased. The authors have also developed an experimental PIM-D constructed with transistor-transistor-logic (TTL) ICs.

In this paper, the detailed architecture of PIM-D is described. The brief description of the target languages and their execution models are given in Section 2 and 3, respectively. The machine architecture is presented in Section 4. The preliminary evaluation results of the machine are discussed in Section 5.

2. TARGET LANGUAGES

The languages to be supported on PIM-D include AND-parallel and OR-parallel Prolog. Both languages are based on Horn logic, a subset of first-order predicate logic. AND-parallel Prolog provides an inter-process communication facility by sharing logical variables among AND processes (goals), while OR-parallel Prolog provides independent solution search among OR processes.

GHC is one of AND-parallel Prolog, such as Concurrent Prolog [15] or PARLOG [6], and was chosen as the basic language of fifth generation computers in ICOT because it has clearer semantics and provides more efficient implementation than Concurrent Prolog and because it has more powerful descriptive power than PARLOG [16].

GHC provides process synchronization among producer and consumer processes using the guard mechanism based on Dijkstra's guarded command [8]. GHC programs consist of guarded Horn clauses. Each clause has the following format:

H :- G1, G2, ..., Gm | B1, B2, ..., Bn.

The symbol '|' is called a commit operator, and the left and right sides of the commit operator are called a guard and body, respectively. H is called a head literal, G1, G2, ..., Gm are called guard literals, and B1, B2, ..., Bn are called body literals. Each clause is invoked when a goal literal is given. Unification is attempted between the head literal and the given goal literal and if it succeeds then the guard literals are invoked as the new goal literals. Only one clause whose guard (i.e., head unification and invocation of all the guard literals) succeeds proceeds its body invocation. That is, one clause is exclusively selected for a given goal from all the clauses whose guards succeed. The selected clause may invoke its body literals as the new goals.

The guard or body literals can be invoked in parallel, and a synchronization function of the processes invoked from these literals is implemented by the guard mechanism. That is, these literals can have shared variables and the invoked clauses may communicate messages via these shared variables. In the guards of the invoked clauses, if unification between a goal variable and a non-variable term is attempted, it is suspended until the goal variable is instantiated to a term (message). Thus, these clauses act as the consumer processes. In the bodies of the invoked clauses, however, unification between a goal variable and a term instantiates the goal variable to the term and will activate the suspended unification operations. Thus, these clauses play the role of producer processes.

On the other hand, a class of all-solution searching problems is easily described in OR-parallel Prolog by using its nondeterminacy. The OR-parallel Prolog programs consist of Horn clauses with no guard mechanism. The emphasis of OR-parallel Prolog is on OR-parallelism rather than on AND-parallelism. If a goal is given, the clauses are invoked and the created OR-processes will search for independent solutions in parallel. The resulting solutions are merged into streams by stream merging primitives and then returned to the goal [12].

3. EXECUTION MODELS

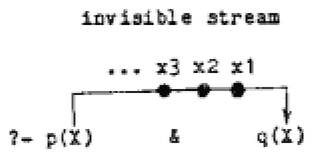
It will be necessary to manipulate large structured data in the knowledge information system to which the fifth generation computers are aiming. Such structured data should be shared among the processing elements rather than being locally copied from a processing element to others because the machine performance may extremely be degraded for the heavy network traffic and because the large redundant storage for the local copies will be required. When the sharing method is adopted, however, the process context switching overhead caused by the frequent remote data accesses will also degrade the performance, if conventional processors were used as the processing elements. The dataflow model assures the independence of the primitive operators and is suitable in such a distributed, parallel processing system.

The machine is based on the tagged token architecture, in which a unique identifier is allocated to each procedure instance [3] [9]. Every procedure instance, therefore, is executed independently as well as the executable nodes in the procedures. Thus parallelism in the programs can easily be exploited by the machine.

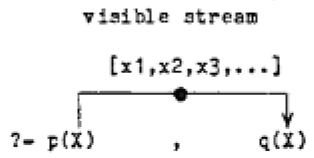
Two types of logic programming languages are supported in a uniform manner; inter-process communication is performed via streams, which are non-strict data structures to facilitate asynchronous communication between consumer and producer processes.

In OR-parallel Prolog, each goal literal execution will produce a set of solutions. These solutions may be returned to the goal in a nondeterminate manner. OR processes will return the solutions to the goal in the order in which the solutions are obtained. These solutions are merged into a stream by stream merging primitives. The stream here is called an invisible stream, because the structure of the stream is 'not visible' to the programmer directly. Figure 1 (a) shows an example of a goal where the instances of variable X obtained by the execution of the literal p(X) are sent to the next literal q(X).

In AND-parallel Prolog, however, the streams are 'visible' to the programmers. The programmers may code the programs that explicitly generate or consume the streams, which are implemented as structured data containing unbound variables as their elements. Such streams are called visible streams. Figure 1 (b) shows an example, where a producer invoked by the literal p(X) generates a visible stream represented by a list and a consumer invoked by the literal q(X) reads its elements. In contrast to OR-parallel Prolog, at most one instance is bound to the variable X and it may be a structure representing a visible stream.



(a) Communication on an Invisible Stream



(b) Communication on a Visible Stream

Fig.1 Inter-process Communication via Streams

The programs written in OR-parallel or AND-parallel Prolog are compiled into dataflow graphs. A dataflow graph is represented by nodes connected by directed arcs. Each node corresponds to an operator and each directed arc corresponds to a data path along which a token carrying the operand for the destined operator is sent. All the nodes whose operands are ready by arrivals of the tokens to their input arcs can be executed in parallel. Each node, when executed, may put new tokens on its output arcs, and thus activate the next nodes.

Figure 2 shows a sample program written in GHC and its compiled dataflow graph code. The program is a list-append program which appends a list specified by the second argument of the head literal to the end of the list specified by the first argument. The appended list is unified with the third argument. In the guards of the clauses, the first goal argument is unified with non-variable terms (nil or list). If the goal argument is an uninstantiated variable, unification is suspended. If instantiated, the succeeding clause will proceed to its body and will instantiate the third goal argument to the second goal argument (the first clause) or a newly created list (the second clause).

The first statement of the compiled code specifies the procedure name and arguments of 'append' procedure. The procedure body is enclosed by 'begin' and 'end' statements. Each body statement corresponds to a node in the dataflow graph, and the left side of the '=' operator specifies destination paths for the results of the instruction specified by the right side of the '=' operator. The '<<=' operator specifies a procedure invocation macro. The dataflow graph representation of the compiled code is shown in Fig. 3.

```
/* GHC source programs */
app([],Y,Z) :- true | Z=Y.
app([H|X],Y,Z) :- true | Z=[H|Z1], app(X,Y,Z1).

/* compiled code */
ret<=app_3(arg1,arg2,arg3).
begin.
    uarg1=wait_instance(arg1).
    (arg1_1,_,arg1_3,_)=switch_by_type(uarg1,uarg1).
    (arg2_1,_,arg2_3,_)=switch_by_type(arg2,uarg1).
    (arg3_1,_,arg3_3,_)=switch_by_type(arg3,uarg1).
    (ret_1,_,ret_3,_)=switch_by_type(ret,uarg1).
    dec_prefc_by_type((1,0,1,0),uarg1).
% body of the first clause.
    e3=write_instance(arg3_1,arg2_1).
    return(e3,ret_1).
% body of the second clause.
    (p4,p5)=decompose_list(arg1_3).
    p7=create_global_var(arg1_3).
    p10=cons_list(p4,p7).
    e11=write_instance(arg3_3,p10).
    e12<<=app_3(p5,arg2_3,p7).
    e13=check_consistency(e11,e12).
    return(e13,ret_3).
end.
```

Fig. 2 The GHC Program and the Compiled Code.

The procedure performs clause indexing at first; a subset of the clauses may be selected by their head or guard information instead of wastefully invoking every clauses. The 'wait_instance' instruction reads the instance of the first goal argument which is passed along the path 'arg1'. If the goal argument is an unbound variable, it is suspended until the variable is instantiated. This operation will need remote access if the variable cells are distributed over the memory units in the system. Then, the 'switch_by_type' instructions switch all the goal arguments according to the first argument; they put their left operands on their first destinations if the first goal argument is 'nil', and on the third destinations if the first goal argument is a list. Thus, one of the clause bodies is invoked exclusively. Because mutual exclusive selection is performed by clause indexing, there is no need to execute the commit operation (i.e., to execute the 'test_and_set' operation on a semaphore flag shared among the clauses) in this example.

The 'write_instance' instruction tries to unify its two operands and if one of them is a variable, it will instantiate the variable to another operand (i.e., the term is written to the variable cell). In the second clause, there is a variable 'Z1' in the body which does not appear in the guard. For such a variable, the 'create_global_var' instruction creates a new variable cell and initialize it. Two body literals in the body will be executed in parallel (one is the 'write_instance' instruction and the other is the recursive invocation of the predicate). The 'check_consistency' instruction tests their results whether they terminated successfully or not.

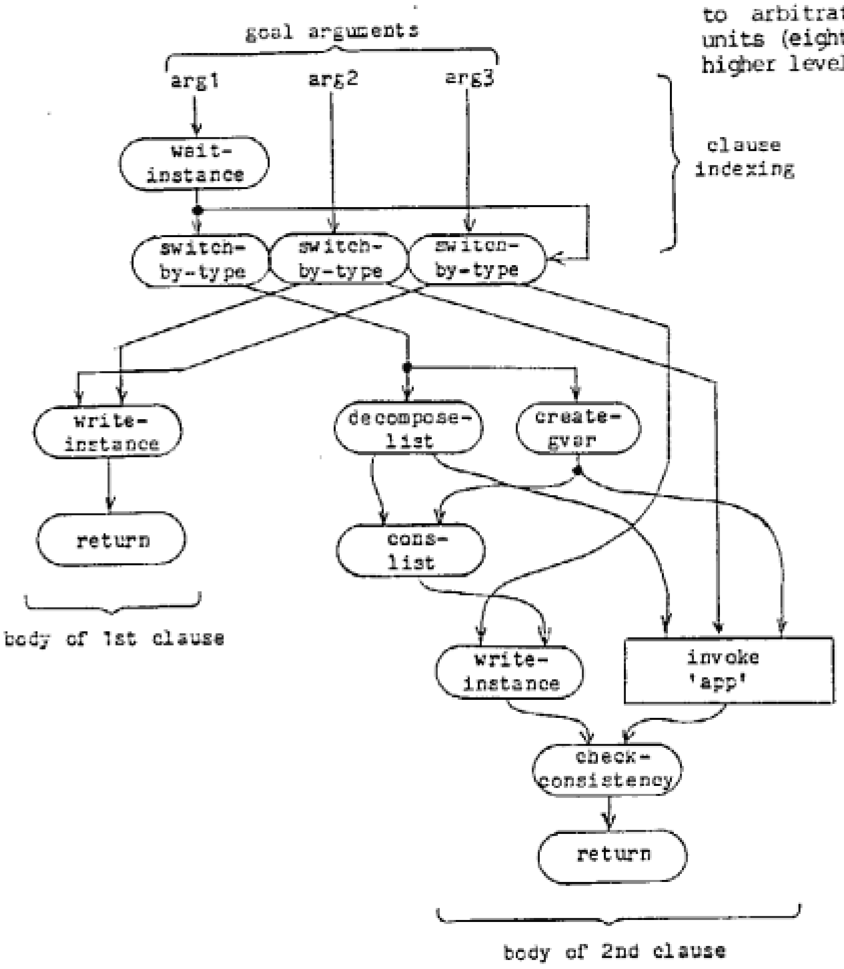


Fig. 3 Dataflow Graph Representation of the Compiled Code

4. MACHINE ARCHITECTURE

The experimental machine is constructed from multiple Processing Element modules (PEs) and multiple Structure Memory modules (SMs) interconnected through a hierarchical network as shown in Fig. 4 [11]. There are several hierarchy levels in the interconnection network. Each PE has its local bus. Four PEs and four SMs are interconnected by an inter-module network bus. A set of these modules is called a cluster. Several clusters are further interconnected by an inter-cluster network bus.

The implemented experimental system consists of two clusters and is currently being expanded to four clusters. Of these clusters, one is specialized whose one SM is replaced by a host processor (VAX-11/730), which is used to initialize or monitor the system. The system can easily be extended to any level of super clusters. The hardware specifications for these interconnection busses are the same, and they are called T-busses. Packet transmission via a T-Bus is controlled by a Network Node (NN), which has a nine-to-one arbiter to arbitrate the requests from its lower level units (eight units at a maximum) and from its higher level bus.

4.1 Packet Formats

Each PE has several stages in order to implement pipelined or parallel execution. Packets transferred between these stages include result packets and executable instruction packets, as shown in Fig. 5. A result packet, or a token which is sent along the directed arc in the dataflow graph, consists of three fields: an activity identifier, a destination, and a data field. The activity identifier (16 bits) specifies the invoked procedure instance name to which the result packet belongs. The destination field (24 bits) specifies the address of the destination instruction (a node in the dataflow graph) of the result packet. It also include two bits for additional information; one specifies whether the destined instruction receives one or two operands (i.e., whether the instruction is executable on the arrival of a single operand or of two operands), and the other is the port number of the destined instruction (i.e., it specifies whether the operand is a left or a right operand). The data field (32 bits) contains the operand data to be sent to the instruction.

The data field is further divided into a tag subfield (7 bits), which specifies the data type such as integer, symbol, list, and so on, and a data value subfield (25 bits). If the data is structured data, the data value field contains a pointer to the structure memory, which consists of a 5-bit module number and a 20-bit local address in memory. Thus, the experimental machine can be expanded to 32 modules (PEs or SMs) with each module able to have up to a one million (20-bit) word memory space.

An executable instruction packet consists of a current instruction address, activity identifier, operation code, left operand, right operand, and destination specifier fields. Of these, the current instruction address indicates the instruction address to be executed and is used to obtain the destination addresses from the destination specifier field as described below. The operation code field (8 bits) specifies the operation to be executed and the destination specifier field (48 bits) specifies the destination addresses of the results. There are two modes to specify the destination addresses in the destination specifier filed; in the full destination mode the specifier field contains up to two destinations (each of them is of 24-bit length), and in the short destination mode the specifier field contains up to four destinations, where each destination is of 12-bit length and contains the relative address from the current instruction. These relative addresses are added to the current instruction address to obtain the absolute addresses.

4.2 Configuration of Processing Element Module

The stages in a PE include a Packet Queue Unit (PQU), an Instruction Control Unit (ICU), several Atomic Processing Units (APUs), and a Network Node (NN), as shown in Fig. 4. These functional units have their own controllers and are operative in a pipelined manner. The PE has a Local Memory Unit (LMU), which is used to store local data such as activity management information, and is shared and accessible from APUs.

The PQU consists of a FIFO (First-In First-Out) queue memory to store result packets (tokens) temporarily. The ICU is microprogram-controlled and has an Operand Memory (OM), Instruction Memory (IM), and hardware hashing logic. The ICU receives the result packet from the PQU and detects the readiness of the instruction operands specified by the arriving result packet, if the instruction receives two operands; if the instruction receives one operand, the instruction is executable and the ICU constructs an executable instruction packet as described below. On a two-operand instruction, the ICU searches associatively the OM, which is used to store the operands whose partner operands are not ready, with the packet's activity identifier and destination fields as a key using the hardware hash. On a successful search, the ICU fetches the partner operand from the OM as well as the instruction code

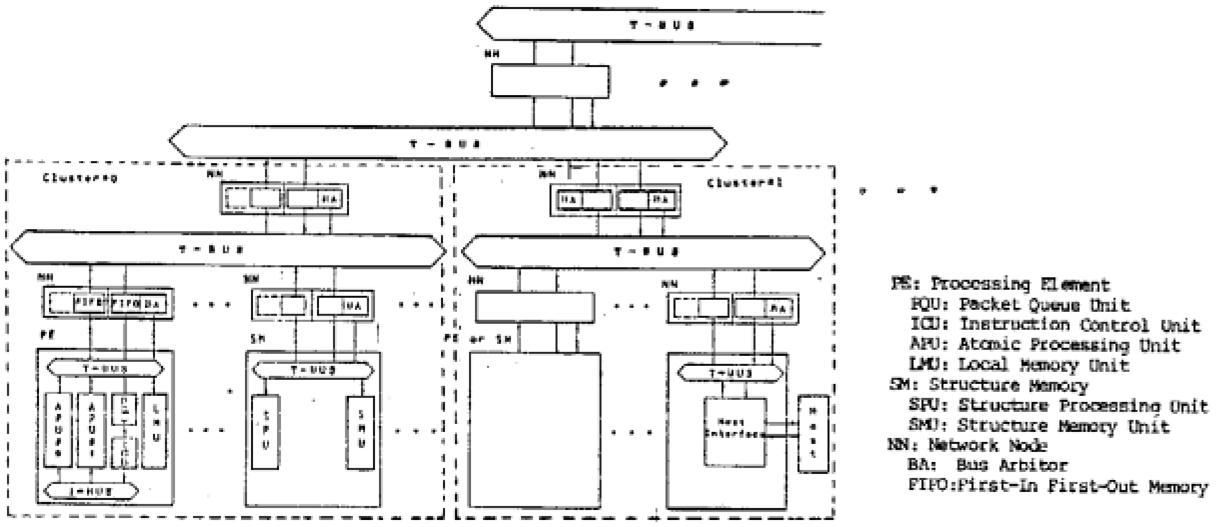


Fig. 4 Configuration of the Experimental Machine

activity identifier (16)	destination address (24)	data (32)

(a) Format of a Result Packet

current address (20)	activity identifier (16)	left operand (32)	right operand (32)	destination specifier (48)

(b) Format of an Executable Instruction Packet

Fig. 5 Packet Formats

from the IM, and constructs an executable instruction packet, which is then sent to the next stage: one of the APUs; otherwise, the ICU stores the arriving operand into the OM until its partner operand arrives.

The executable instruction packet is sent to one of the APUs, which are also microprogram-controlled with AM-2900 series bit-sliced microprocessors as the arithmetic logic units. The cycle time of the APUs is 333 nano seconds. The APU performs computation specified by the instruction code and generates new result packets to be sent to the FQU or other PEs. If structure accesses are necessary, the structure manipulation commands are sent to the SMs pointed to by the structure pointer fields in the operands. In the experimental system, two APUs are implemented in one PE.

The high-bandwidth data paths were adopted for communication between the functional units. As shown in Fig. 5, the size of a result packet is 72(=16+24+32) bits. The T-bus is wide enough to send a result packet from one unit to another in one T-bus transmission cycle (500 nano seconds). The bus connecting the ICU and the APU, which is called an I-bus, has a 48-bit width; an executable instruction packet is sent by four I-bus cycles (166.7 X 4 = 667 nano seconds).

4.3 Configuration of the Structure Memory Module

Each SM consists of an SPU (Structure Processing Unit) and Structure Memory Unit (SMU) for storing the structured data. The SPUs receive the structure manipulation commands from the APUs and interpret them. If the commands need the responses, new result packets are created and sent back to the PEs. Such commands include read commands, memory allocation commands, and so on.

Synchronization via shared variables are done by additional tag fields in the structure memory [4]. Each memory cell word in memory has a tag field specifying whether the contents of the memory word are valid or not. The "empty" value indicates that the word is empty (i.e., no write operation to the word has been performed yet). The "pending" value indicates that some read operations have been performed to the empty word (the read operations are suspended and the suspended read requests are chained into the memory word until the write operation to the word is performed). Other tag values indicate the data type of the data written into the word.

Garbage collection of structure memory is performed by the reference counting method [1] [2]. 10-bit reference count fields are appended to every two word cells, because the basic cell in PIM-D is the list cell, which consists of two contiguous words. The reference count has the number of pointers which point to the cell, and if the reference count reaches to zero the cell is reclaimed.

The specification of the various units is shown in Table 1.

Table 1. Specification of the Units

unit		specification
PE	FQU	FIFO size: 16 K words X 86 bits (16 K tokens)
		cycle time: 167 nano seconds
	ICU	IM size: 96 K words X 59 bits (96 K instructions)
		OM size: 32 K words X 64 bits (32 K tokens)
		cycle time: 333 nano seconds
	APU	micro store:1 K words X 32 bits Read Only Memory
		7 K words X 32 bits Random Access Memory
		cycle time: 333 nano seconds
	LMU	memory size:512 K words X 32 bits
SM	SPU	micro store:1 K words X 32 bits Read Only Memory
		7 K words X 32 bits Random Access Memory
		cycle time: 333 nano seconds
	SMU	memory size:1024 K words X 34 bits (for data and tags)
		512 K words X 10 bits (for reference count)
NN		FIFO size: 64 words X 86 bits (64 tokens)
		cycle time: 167 nano seconds

Two versions of the firmware interpreters are being implemented; one is an SM-distribution system and the other is an SM-integration system. In the SM-distribution system the structured data is distributed to and stored in the SMs. Each APU, therefore, sends the structure manipulation commands to the SMs, if structure access is necessary.

In the SM-integration system, however, the structured data is distributed to and stored in the local memories of the PEs. In this system, the amount of hardware logic is reduced to about two-third of the SM-distribution system because no SMs are necessary. Each APU can access its local memory if the structured data is stored in the local memory, rather than sending the structure access command to other modules. In this case, the locality of structure access should be necessary in order that many structure accesses may not cause the APUs to send the structure commands to other PEs. If there are many remote structure accesses, the load of the APUs become heavier than the SM-distribution system, because the APUs must process these commands as well as their instruction execution.

4.4 Process Allocation

We have chosen a process allocation strategy, which can exploit parallelism by distributing the active processes, or procedures, among clusters or PEs while making use of the locality of the programs. Process allocation is performed by a procedure invocation instruction. The APU, if it receives this instruction as an executable instruction, allocates one of the PEs to the new procedure by testing the process distribution factor. The process distribution factor is decided by the load status of all the PEs; according to the network hierarchy levels of the system configuration, there are roughly three levels of distribution factor. It is assumed that the programs to be executed have large amount of parallelism and that the degree of parallelism may dynamically be changed. If the system load is light (i.e., if the degree of parallelism is low as in the initial execution stage), the distribution factor is high and the new procedures are distributed over the clusters, if the load becomes intermediate, the new procedures may be allocated to one of four PEs in the cluster, and if the load is heavy, the distribution factor is low and the new procedures may be allocated to their own PEs.

In order to maintain the distribution factor, each PE may transfer the load status such as the packet queue length to other PEs, when some specific time period has expired, or when the queue length exceeds some threshold level. This transmission may be performed at first between the four PEs in the cluster and if a drastic change of the cluster's load status occurs from the previous one, then the status may be transmitted to the other clusters.

This load balancing scheme is like that of AMPS [10] except that the user or programmer can specify the process allocation strategy. There are three types of procedure invocation instructions according to the hierarchy levels: an inter-cluster call, intra-cluster call, and internal call instructions. The inter-cluster call instruction follows to the above scheme; this instruction is used for distribution of loosely-coupled processes. The intra-cluster call instruction allocates one of the PEs in the cluster to the new procedure when the distribution factor is high; otherwise, it allocates the own PE. The last one, internal call instruction, always allocates the own PE to the new procedure, thus the local computation is assured. This instruction, for example, is used for the tail recursive calls or for allocation of the tightly-coupled processes.

5. EVALUATION RESULTS

The first version of the SM-distribution system was microcoded and various evaluation results have been obtained. The evaluated sample programs include:

- N-queens program to find all solutions to place the N queens on the N X N chess board so that no queen captures any other queen (written in OR-parallel Prolog and GHC),
- BUP (Bottom Up Parser) program that analyzes a simple Japanese sentence (written in OR-parallel Prolog),
- quick-sort program that sorts a list of 255 elements in ascending order (written in GHC).

They are compiled into the dataflow graphs as shown in Fig. 2 and initially loaded into the ICUs of the machine from the host processor. The Programs are duplicated in all PEs.

Figure 6 shows the total machine cycles needed to execute the programs and the performance of the experimental machine when the number of modules (PEs and SMs) is increased. The performance of the machine with single PE and SM is about 2.5 or 3.2 K RPS (Reductions Per Second), where the number of reductions means the number of successfully terminated goals. It is not so high because

- the vertical microprogram format is adopted rather than a horizontal format to simplify the hardware,
- the internal bus of the APU or SPU is of 32-bit width, hence several micro steps are needed to create a packet,
- no microprogram optimization has been done yet.

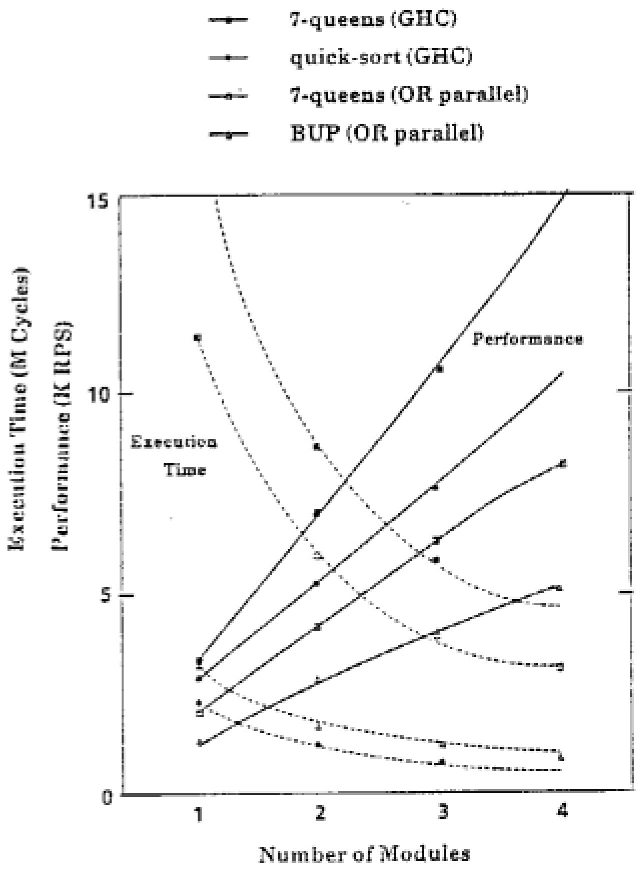


Fig.6 Performance of the Experimental Machine

The performance, however, is increased linearly when the number of modules is increased even if the structured data is distributed over the SMs, and will be improved by firmware optimization and hardware refinement. In order to extend the machine to the prototype system constructed from hundreds of processors, the authors are investigating a VLSI version of PIM-D and estimated the improved performance of the new PIM-D. The main bottle-neck in the current experimental machine is in the APU as described above, and the APU's machine cycles needed to interpret the instructions may be reduced to one-third or one-fourth by extending the internal bus bandwidth. The cycle time will be improved to half or one-third by using the VLSI technology. Finally, the number of APUs in a PE is extended to six or eight from two. Thus, the improved performance is potentially in the range from 18 to 48 times faster than the experimental machine (it is in the range from 54 to 144 KRPS). As described in Section 3, it is necessary that the large structured data should be distributed and shared among processing elements. It was shown that the dataflow machine is suitable for such a distributed, parallel processing system.

6. CONCLUSION

Execution models on the dataflow-based parallel inference machine for OR-parallel and AND-parallel Prolog and the experimental machine architecture were described. It was shown that two types of logic programming languages with different aims can be supported on this machine. The programs are compiled into dataflow graphs corresponding to machine language codes. Thus, parallelism in the programs can be exploited naturally.

The machine is constructed from processing elements and structure memories interconnected through a hierarchical network. The processing elements interpret the procedures represented by the dataflow graphs in parallel. Structured data is distributed to structure memories and shared among these procedures.

Detailed designs for the experimental machine have been developed and the first version of the firmware interpreter debugged. The preliminary evaluation results of OR-parallel and GHC programs indicate that performance is linearly improved by exploiting parallelism. Future efforts will involve optimization of the firmware interpreter and improvement of the machine hardware to serve as the basis for a highly parallel inference machine.

<Acknowledgments>

The authors extend their thanks to Director Kazuhiro Fuchi at ICOT, who afforded them the opportunity to pursue this research. Also much appreciated are Dr. Shun-ichi Uchida, Chief of the Fourth Research Laboratory, and Dr. Atsuhiro Goto (ICOT) for their valuable advice and comments, Mr. Toshihiko Miyazaki (ICOT), for his implementation of the compiler, Teruhiko Oohara and Masayuki Tomisawa (OKI) for their hardware debugging, and other ICOT research members for their fruitful discussion.

<References>

[1] Ackerman,W.B., " A Structured Processing Facility for Data Flow Computers," Proceeding of International Conference on Parallel Processing, 1978.

[2] Amamiya,M., R.Hasegawa, O.Nakamura, and H.Mikami, "A List-processing oriented Data Flow Architecture," National Computer Conference 1982, pp. 143-151, June, 1982.

[3] Arvind, K.P.Gostelow, and W.E.Plouffe, "An Asynchronous Programming Language and Computing Machine," TR-114a, Dept. of ICS, University of California, Irvine, Dec., 1978.

[4] Arvind and R.E.Thomas, "I-Structures: An Efficient Data Type for Functional Languages", TM-118, Laboratory of Computer Science, MIT, 1980.

[5] Arvind and R.A.Innucci, "A Critique of Multiprocessing von Neumann Style," Proceedings of 10th Internatinal Symposium on Computer Architecture, June, 1983.

[6] Clark,K. and S.Gregory, "PARLOG: Parallel Programming in Prolog," Research Report DOC 84/4, Imperial College of Science and Technology, April, 1984.

[7] Dennis,J.B. and D.P.Misnus, "A Preliminary Architecture for A Basic Data Flow Processor," Proc. of 2nd Symp. on Computer Architecture, Jan., 1975.

[8] Dijkstra,E.M., "A Discipline of Programming," Prentice-Hall, 1976.

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

[10] Keller,R.M., G.Lindstrom, and S.Patil, "A Loosly Coupled Applicative Multi-processing System", AFIPS Conference Proceedings 48, June, 1979, pp.613-622.

[11] Kishi,M., E.Kuno, K.Rokusawa, and N.Ito., "Architecture of Experimental System of Dataflow-based Parallel Inference Machine," Proc. of 30th National Conference of Information Processing Society in Japan, 1985 (in Japanese).

[12] Ito,N. and K.Masuda, "Parallel Inference Machine Based on the Data Flow Model," International Workshop on High Level Computer Architecture 84, Los Angeles, California, May, 1984.

[13] Ito,N., H.Shimizu, M.Kishi, E.Kuno, and K.Rokusawa, "Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog," New Generation Computing, Vol. 3, No. 1, 1985.

[14] Ito,N., M.Kishi, E.Kuno, and K.Rokusawa, "The Dataflow-Based Parallel Inference Machine To Support Two Basic Languages in KL1," Proc. of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, UMIST (Manchester), July 1985.

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

[16] Ueda,K., "Guarded Horn Clauses," TR-103, Institute for New Generation Computer Technology, Tokyo, Japan, 1985.