DATA FLOW BASED PARALLEL INFERENCE MACHINE

UDK 681.3.001:519.6

Jurij Šilc and Borut Robič
Jožef Stefan Institute, Ljubljana

Abstract. Execution models of the data flow based parallel inference machine for OR-parallel and AND-parallel Prolog and the experimental machine architecture are presented. It is shown that two types of logic programming languages with different aims can be supported on this machine. The programs are compiled into data flow program graphs corresponding to machine language codes. Thus, parallelism in the program 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 data flow program graphs in parallel. Structured data is distributed to structure memories and shared among these procedures.

Keywords. Fifth generation computer systems, parallel inference machine, OR-parallel Prolog, AND-parallel Prolog, data flow mechanism, machine architecture.

1 Introduction

The Fifth Generation Computer Systems (FGCS) research and development aim is to build a prototype of knowledge information processing system capable of efficiently performing knowledge-based problem solving and inference. Toward this end, a ten-year period has been assigned to the FGCS Project, and this period has been further devided into three stages. The goal of the initial three-year stage is to conduct basic research on individual system components in order to establish basic configuration technology for subsystems which are to be realized in the intermediate four-year stage.

Fig. 1 shows what has become known as the "basic configuration image" of the fifth generation computer [1]. Looked at vertically, it has a hardware layer, a software layer, and an external interface to applications systems, as might be expected. Looked at horizontally, it becomes clear that each aspect of the functionality of a fifth generation computer - problem solving and inference, knowledge base management and intelligent interfacing - requires its own hardware and software support mechanisms.

The parallel inference machine and knowledge base machine are the most important hardware components of the FGCS. In the FGCS prototype to be completed as the final product of the project, the two machines will be integrated through a close link. In the initial stage, however, research and development are proceeding separately for each machine with research themes separately determined, since the initial stage mainly aims to conduct research and development of individual component technologies to establish the basic technology for the hardware, called the inference subsystem and knowledge base subsystem to be build in the intermediate stage [8].

2 Knowledge base and inference subsystems

Development of the FGCS hardware and architecture will include the implementation of mechanisms for processing and controlling a knowledge base and efficient execution of problem-solving and inference techniques with these mechanisms. The system will depend on multiprocessing and parallel processing techniques for which two objectives are critical:

(1) Provide machines with the power to handle the natural parallelism found in problems tackled by humans. The structure of a problem and the neccessary processing for solving it can be shown by rules controlled by an inference mechanism. Thus a major goal of the FGCS project is to devise an execution model for the inference mechanism and to determine a way to configure it.

(2) Achieve high-speed parallel processing capable of supporting intelligent human activities. For this requirement, the principal research must be concentrated on knowledge base processing algorithm to handle a large number of facts as well as a mechanism supporting the algorithm.

The inference subsystems, together with the knowledge base subsystem, forms the kernel of the FGCS hardware [8]. The ultimate aim of the FGCS research and development project is a machine enabling the execution of parallel inferences [3,7,9]. In the following we shall describe some FGCS project "data flow directed" efforts in designing such a machine.

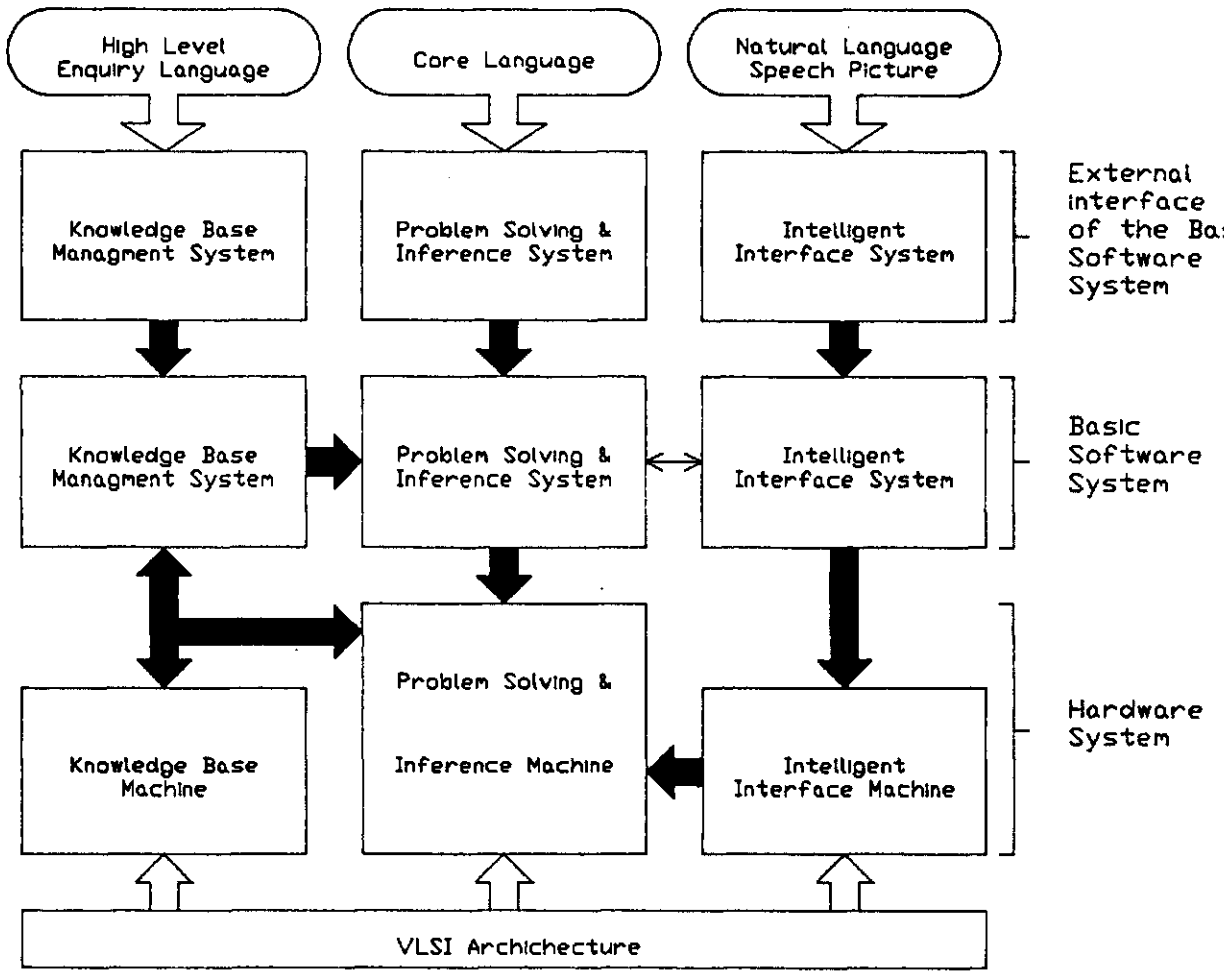

High Level Enquiry Language
Core Language
Natural Language Speech Picture
Knowledge Base Managment System
Problem Solving & Inference System
Intelligent Interface System
External interface of the Basic Software System
Knowledge Base Managment System
Problem Solving & Inference System
Intelligent Interface System
Basic Software System
Problem Solving & Inference Machine
Knowledge Base Machine
Intelligent Interface Machine
Hardware System
VLSI Archichecture

Fig. 1 The overall structure of a fifth generation computer.

3 Parallel inference subsystem

Machine language of parallel inference machine

In FGCS project, logic programming was selected as a bridge to fill the gap between a highly parallel computer architecture and knowledge information processing. Several logic programming languages, named kernel languages, which define an abstract interface between the hardware and the software, are beeing developed. The kernal language KL0 is the machine language of sequential inference machine from which a parallel version KL1 is beeing developed. KL1 is the machine language of parallel inference machine [2] including two types of basic languages: AND-parallel Prolog and OR-parallel Prolog [4].

In the execution of logic programs, a high degree of parallelism can be implemented with use of AND-parallel and OR-parallel executions. When OR-parallelism is applied alternative clouses of the same goal are executed in parallel. The alternative clauses have identical initial states and do not interfere with each other, except for possible concurrent initialization attempts of a goal variable by multiple clauses. On the other hand, in AND-parallelism the conjunctive goals of a clause body are executed in parallel. In general, the goals may share variables and thus interfere with each other when the shared variables are accessed concurrently. AND-parallelism in logic programming involves the simultaneuos execution of subgoals in a clause. Whereas OR-parallelism attempts to achieve increased speed by investigating many possible solutions in parallel, AND-parallelism attempts to achieve increased speed by investigating the subparts of a particular solution in parallel.

In conventional sequential Prolog the search and test operations (called unifications) are executed one by one, but parallel search and test operations can be implemented through parallel machine architecture to obtain a high-speed machine. A few sources of parallelism which can be distinguished for parallel execution of Prolog are AND parallelism, and OR parallelism.

(1) OR-parallel Prolog. When a goal literal G is given, the definition of G is invoked. A clause C is then selected from the definition, and unification of G and the head H of the clause C is attempted. Generally, when multiple clauses C1, C2, ... Cn exist in the definition, unification of G and each H1, H2, ... Hn can be executed in parallel. A unit clause Ci that is successfully unified with G returns the solution(s). A nonunit clause Cj initiates the next unification, treating its body as a new goal statement, and waits for the solutions. The resulting solutions of the goal G are merged into streams by stream merging primitives (in the order in which they are obtained) and then returned to the goal. Thus, OR-parallel Prolog is suitable for the clas of "search-for-all-solutions" problems.

An example of OR-parallel Prolog is Parallel Prolog [4].

(2) AND-parallel Prolog. When a goal is expressed as G1 AND G2 AND ... AND Gm, AND-parallelism can be used to search for conditions for all literals Gi in parallel. The goal statement is satisfied only when solutions are found for all the literals Gi and there is no inconsistency between these solutions. The consistency checking is easy or even unnecessary in cases when the goal literals Gi have no unbound variables shared among AND processes, or where shared variables are bound to the ground instances before invocations of these literals.

Several languages have been proposed to realize AND-parallelism. They include PARLOG, Concurrent Prolog, and GHC (Guarded Horn Clauses) [4,5].

Mechanisms of parallel inference

Various mechanisms of parallel inference and architectures based on those mechanisms are beeing studied: data flow mechanism, reduction mechanism, complete-copying mechanism and, clause unit processing mechanism. In what follows we shall briefly describe these four mechanisms [8]:

(1) Data flow mechanism. In the data flow concept, execution starts when data necessary for the execution arrived. This concept can result in parallelism regardless of whether it is explicitly indicated in the program. This mechanism executes kernel language programs in parallel based on the data flow concept.

(2) Reduction mechanism. When executed, an OR-parallel and AND-parallel Prolog program generates resolvents from a goal and clause. This can be regarded as a process in which a goal modifies itself using a clause as a rule. The reduction mechanism can also be viewed as a kind of self-modification. Thus, there is a close similarity between the execution of OR-parallel and AND-parallel Prolog programs and the reduction mechanism. Accordingly, the reduction mechanism was selected for a machine architecture that executes OR-parallel and AND-parallel Prolog programs.

(3) Complete-copying mechanism. Complete-copying is type of reduction mechanism. Even if a process includes several literals (subgoals) and only one literal (subgoal) is reducible, the whole process is copied and transferred to a unit that executes the unification process. This increases the number of copies and the length of a packet in the network, while enhancing the independence of each process.

(4) Clause unit processing mechanism. In response to a request from an idle processing unit, a busy processing unit sends a process. Thus, this mechanism can avoid an explosion of resource requests. However, it takes time for all processing units to become busy.

4 Data flow based inference machine

The parallel inference machine based on data flow mechanism (PIM-D) is naturally well suited to parallel procesing becouse the data flow mechanism is closely related to functional languages.

Data flow computation ...

Programs in the data flow model are represented by data flow graphs, nodes correspond to operators and directed arcs correspond to data paths along which operands are sent. An operator is driven by operand arrivals from its input arcs, and it outputs the result operands to its output arcs without affecting the other operators' execution. This functionality of operators has close similarity to the functional languages.

... and logic programming

Execution of logic programs is performed in a goal-driven manner: a clause in the programs is initiated when a goal is given and returns the solutions to the goal. Logic programming languages make use of the unification operation, which is one of their basic functions. Nondeterminism is another basic feature of these languages; in particular, "don't-know nondeterminism" is required for OR-parallel Prolog, while "don't-care nondeterminism" is required for AND-parallel Prolog. The data flow model is also similar to logic programming languages such as OR-parallel and AND-parallel Prolog. The pograms written in OR-parallel or AND-parallel Prolog are compiled into data flow graphs.

Implementation of GHC

GHC was selected as a basic language of KL1 becouse it has clearer semantics and provides more efficient implementation than Concurrent Prolog, and it has more powerful descriptive power than PARLOG. GHC programs consist of guarded clauses such as:

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

where, H, Gi, and Bj are head, guard and body literals, respectively and "|" is called a commit operator. When a goal literal is given each definition clause is invoked and a semaphore flag shared among these clauses is created. 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 the clause whose guard literals succed first can execute its body; i.e. the clause whose guard succeeds performs a test-and-set operation to the shared semaphore flag. If the result of this operation is also successful, the clause can execute its body; processing of the other clauses is terminated. Thus, one clause is exclusively selected for a given goal from all the clauses whose guards succeeded. There are several implementation schemes to support the guard mechanism in GHC [6]:

(1) Complete compilation scheme. All the unification directions are analysed in compilation time and codes are generated using unidirectional unification primitives. In this scheme the compiler is complicated.

(2) System number scheme. All the environments are managed by guard system numbers. A new guard system number is allocated, when a new definition is invoked and is restored to its parent number when the commit operator is executed. The guard numbers are associated with all the variables included in the invoked clauses and the inironment to which each variable belongs is compared with the current environment when unification to the variable is attempted.

(3) Pointer coloring scheme. The pointer coloring scheme distinguishes variables belonging to the goal literals from those belonging to the current guard by coloring.If unification is attempted between a goal variable and a variable in the invoked clause, the callee's variable is changed to a colored variable, which points to the original variables. If a colored variable is unified with a term, the instance bound to the variable is read before unification. The commit operator restores the colored variables to their original variables.

(4) Read-only tagging scheme. This scheme is an extension of the pointer coloring scheme, in which every variable has a tag specifying its read-only level. The read-only levels of the goal variables are incremented by one before the definitions are invoked, and decremented by one when the commit operator of each invoked clause succeeds.

Translation from GHC program into the data flow graph

We shall illustate the translation from a given GHC program into the corresponding data flow graph. Let us have a sample program written in GHC (Fig. 2). It 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 resulting list is unified with the third argument. The compiled code is depicted in Fig. 3. The first statement of the compiled code specifies the procedure name "app" and its arguments "arg1", "arg2", and "arg3". The pocedure body is enclosed by "begin" and "end" statements and consists of three segments. The second and the third segment are compiled codes of the first and second source clause, respectively. The role of the first segment is to decide which of the subsequent segments should be invoked. Each body statement corresponds to a node in the data flow graph. The left side of the "=" operator specifies destination paths for the results of the instuction specified by the right side of the "=" operator. The "<<=" operator specifies a procedure "app" invocation macro. The "wait_instance" instruction reads the instance of the first goal argument which is passed along the input path "arg1". If the goal argument is an unbound variable, it is suspended until the variable is instantiated ("uarg1"). 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 "uarg1". If the first argument is nil, they put their left operands on their first destinations, otherwise (if "uarg1" is a list) they put their left operands on the second destinations. Thus, one of the subsequent segments is invoked exclusively. 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. In the second clause of the source program, 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 initializes it. Two body literals in the body of the second source clause will be executed in parallel. The first 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. The data flow program graph representation of the compiled code is shown in Fig. 4.

```
( GHC SOURCE PROGRAM )

apnd([],Y,Z):-true|Z=Y.
apnd([H|X],Y,Z):-true|Z=[H|Z1],apnd(X,Y,Z).
```

Fig. 2 GHC source program.

```
( COMPILED CODE )

ret<<=app(arg1,arg2,arg3).
begin.
 ( CLAUSE INDEXING )
 uarg1=wait_instance(arg1).
 (arg1_L,arg1_R)=switch_by_type(uarg1,uarg1).
 (arg2_L,arg2_R)=switch_by_type(arg2,uarg1).
 (arg3_L,arg3_R)=switch_by_type(arg3,uarg1).
 (ret_1,ret_2)=switch_by_type(ret,uarg1).
 ( COMPILED CODE OF THE FIRST CLAUSE )
 res1=write_instance(arg3_L,arg2_L).
 return(res1,ret_1).
 ( COMPILED CODE OF THE SECOND CLAUSE )
 (p1,p2)=decompose_list(arg1_R).
 p3=create_global_var(arg1_R).
 p4=cons_list(p1,p3).
 p5=write_instance(arg3_R,p4).
 p6<<=app(p2,arg2_R,p3).
 res2=check_consistency(p5,p6).
 return(res2,ret_2).
end.
```

Fig. 3 Compiled code.

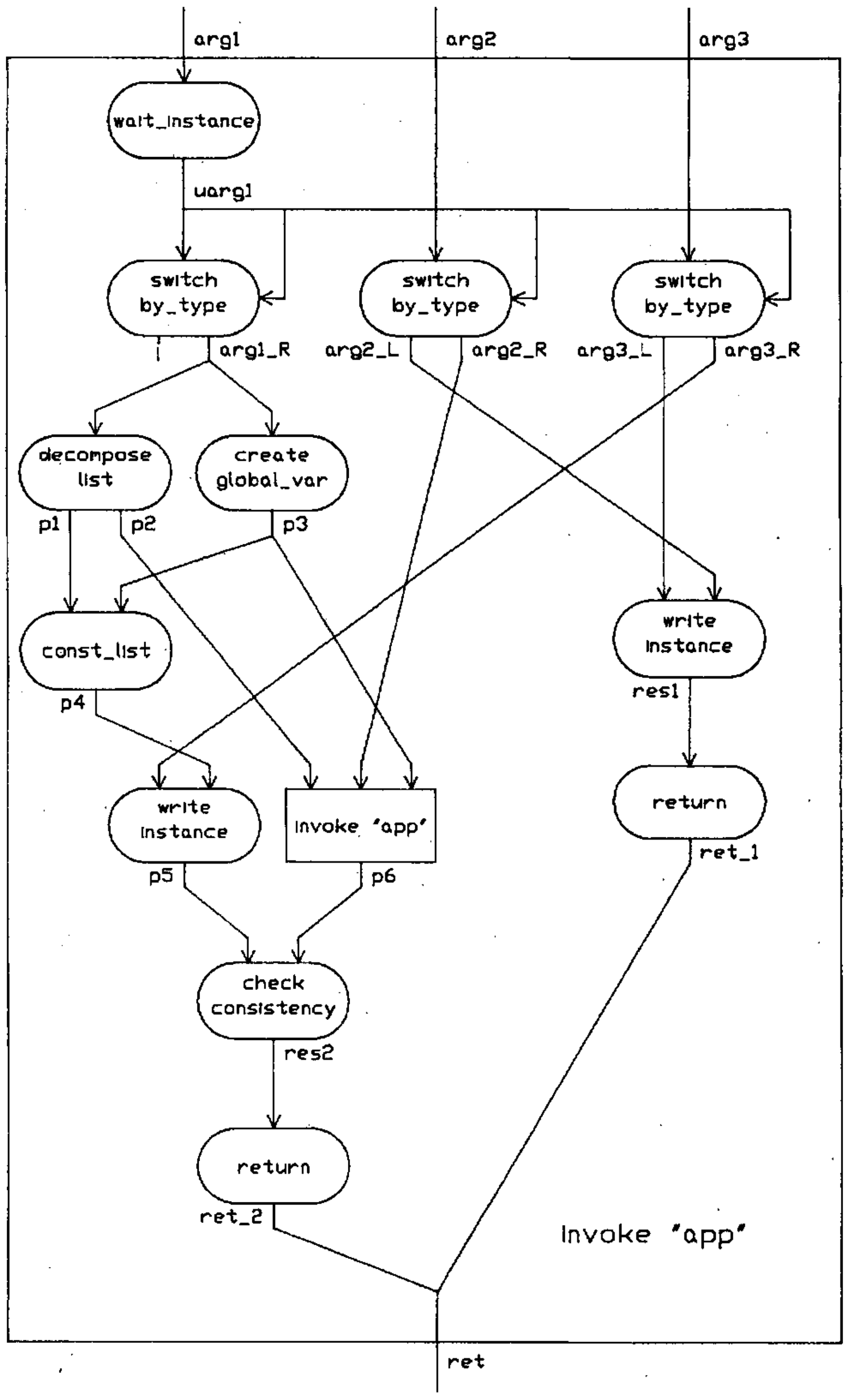


Fig. 4 Data flow graph representation of the compiled code.

Machine architecture

Abstract machine architecture. The machine can exploit OR and AND-parallelism as well as parallelism in unification. In head unification, if both literals consist of multiple arguments, or if both arguments are structured data, the unification of these arguments or their substructures can be executed in parallel. The machine is constructed from multiple processing elements and multiple structure memories interconnected by networks. The abstract machine architecture is shown by Fig. 5 [4].

Experimental machine. The experimental machine is constructed from multiple processing element modules (PEs) and multiple structure memory modules (SMs) interconnected through a hierarchial network as shown in Fig. 6 [5,6]. There are several hierarchy levels in the

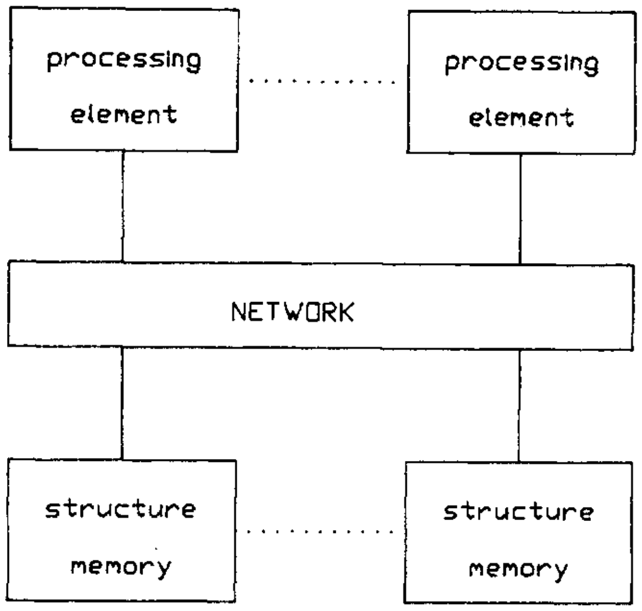


Fig. 5 The abstract machine architecture.

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 futher interconnected by an inter-cluster network bus. The hardware specification for these interconnection busses are the same, and they are called T-busses (token busses).

Actual implementation of the experimental machine includes two clusters and is currently being expanded to four clusters. Of these clusters, one is specialized, having one SM repleced by a host processor (VAX-11/730), which is used to initialize or monitor the system.

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.

A resul packet (a token which is sent along the directed arc in the program graph), consists of three fields:

(1) The activity identifier (16 bits) specifies the invoked procedure instance name to which the result packet belong.

(2) The destination field (24 bits) specifies the address of the destination instruction (a node in the data flow graph) of the result packets. It also includes two bits for additional information; one specifies whether the destined instruction receives one or two operands, and the other specifies whether the operand is a left or right operand.

(3) The data field (32 bits) contains the operand data to be send to the instruction. The machine uses a tagging scheme, in which each operand has a value field (25 bits) and tag field (7 bits), which specifies the data type of the operand. If the operand is a structured data, the value field has a pointer to the structure memory (5-bit module number and a 20-bit local address in memory), and tag field is further divided into two subfield: a data type subfield, which specifies the data type of structure (i.e., list, vector, ...) and a attribute subfield. The attribute subfield contains a non-ground flag, which indicates whether the structure has any simple variables. The attribute subfield also contains a shared flag, which indicates whether the structure has any shared-type variables (i.e., shared variables, global variables, or read-only variables). The machine recognizes the tag field of the operand and transfers control to the appropriate firmware routine.

An executable instruction packet consists of five fields:

(1) The current instruction address (20 bits) indicates the instruction address to be executed and is used to obtain the destination

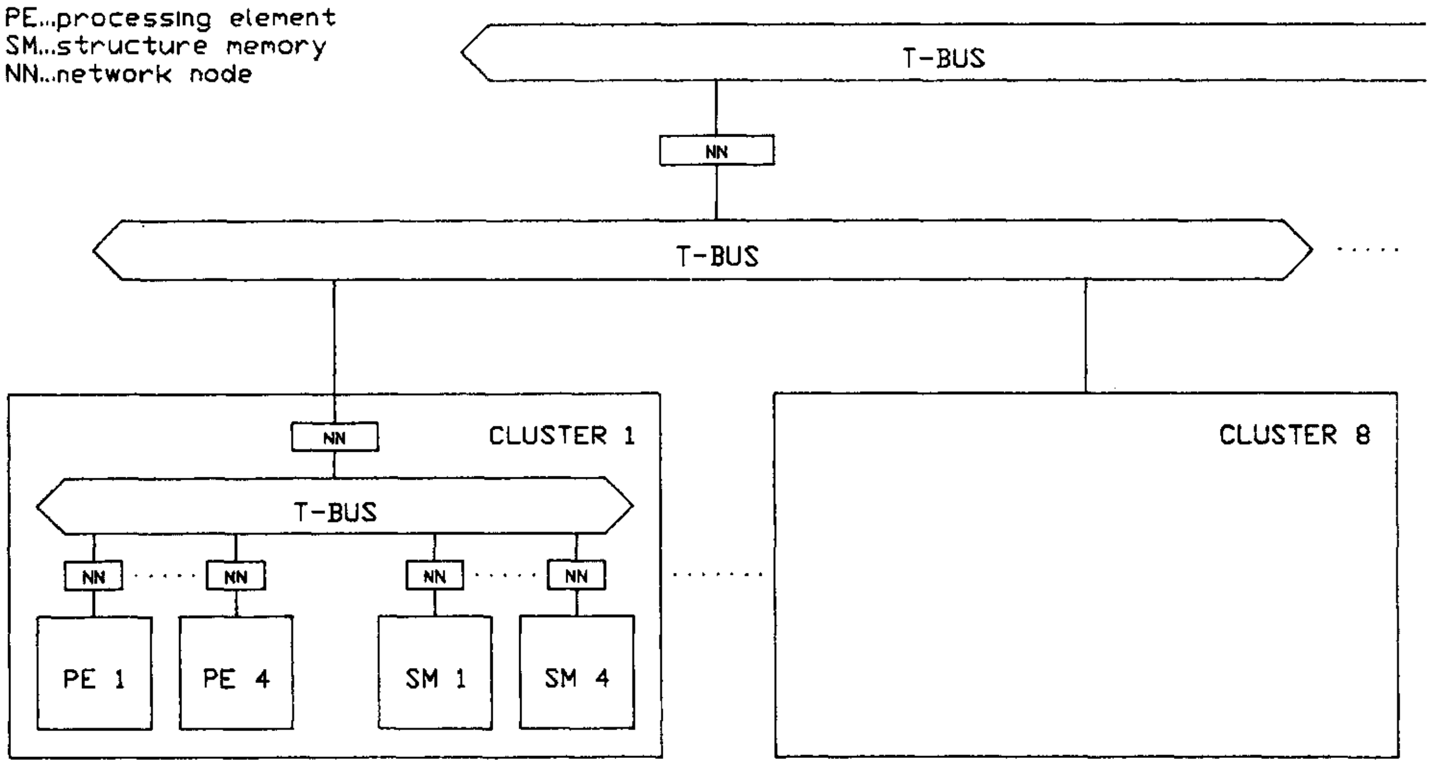


Fig. 6 Configuration of the experimental machine.

address from the destination specifier field as described below.

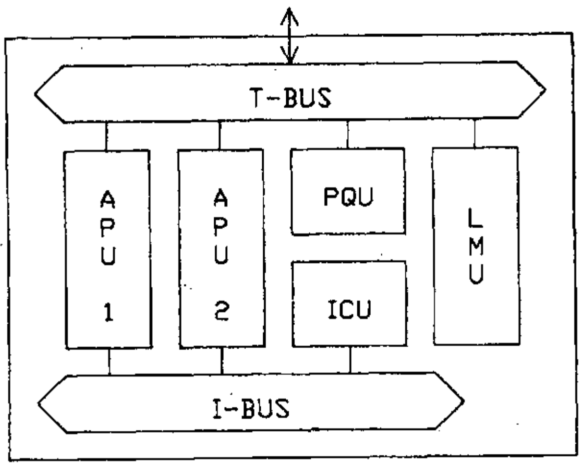
(2) The operation code field (8 bits) specifies the operation to be executed.

(3) The left operand (32 bits).

(4) The right operand (32 bits).

(5) 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 field; in the full destination mode the specifier field contains up to two destinations (each of them is 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 addresses from the current instruction. The relative addresses are added to the current instruction address to obtain the absolute addresses.

Processing element module. Fig. 7 depicts the configuration of each PE. The stages in a PE include a packed queue unit (PQU), an instruction control unit (ICU), several atomic processing units (APUs), and a network node (NN). These functional units have their own controllers and are operative in a pipelined manner. Packet transmission via T-bus is controlled by a NN, which has nine-to-one arbiter to arbitrate the requests from its lower level units and from its higher level bus. The PE has a local memory unit (LMU), which is used to store local data such as activity manegement information, and is shared and accessible from APUs. PQU is a FIFO queue memory to store the result packets from the T-bus. ICU receives the result packets from PQU and checks if the destination instructions are executable or not. An instruction is executable if it receives a

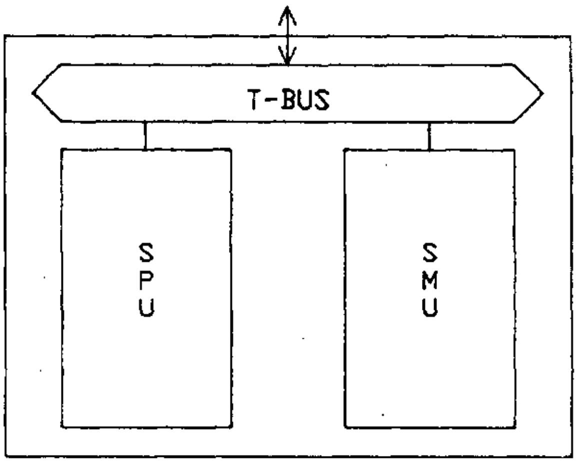


APU...atomic processing unit
PQU...packet queue unit
ICU...instruction control unit
LMU...local memory unit
T-BUS...token bus
I-BUS...instruction bus

Fig. 7 Configuration of a processing element.

single operand, or if the partner operand is already in the operand memory (OM) in the ICU when it receives two operands. In the later case, the ICU searches in its OM whether the partner operand exists or not. If it does, the partner is removed from the memory; otherwise, the result packet is stored in the OM. This searching is performed associatively by hardware hash using the identifier and the destination address as the key field. If the instruction is executable, the ICU fetches the instruction code in its instruction memory (IM) and constructs an executable instruction packet and sends the packet to the next stage, one of APUs via the instruction bus (I-bus). The APU interprets the instruction packets and sends result packets to the PQU in its PE or other PEs, or sends structure access comand packets to SMs via the token bus.

Structure memory module. The SMs are responsible for the structure access commands, perform structure manipulation operations, and return results to the destination specified by the commands. Each SM consists of an structure processing unit (SPU) and structure memory unit (SMU) for storing the structured data (Fig. 8). 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.



SPU...structure processing unit
SMU...structure memory unit
T-BUS...token bus

Fig. 8 Configuration of a structure memory.

The specification and typical processing times of the various units are given in Table 1 and Table 2, respectively.

Table 1 Specification of the units.

unit		specification*
PE	PQU	FIFO size: 16Kw x 86b (16K tokens)
	ICU	IM size: 96Kw x 59b (96K instr.)
		OM size: 32Kw x 64b (32K tokens)
	APU	micro store: 1Kw x 32b ROM
		7Kw x 32b RAM
	LMU	memory size: 512Kw x 32b
SM	SPU	micro store: 1Kw x 32b ROM
		7Kw x 32b RAM
	SMU	memory size: 1Mw x 34b (data, tags)
		512Kw x 10b (ref. count)
NN		FIFO size: 64w x 86b (64 tokens)

* w .. word b .. bit

Table 2 Typical processing times of the units.

unit		item	time*
	PQU	packet receive	2
		delay in queue	8
	ICU	single operand instruction	2
		two operand instruction	
PE		(on arrival of 1st operand)	3
		two operand instruction	
		(on arrival of 2nd operand)	5
	APU	"copy" instruction	3
		packet creation	2
SM		SM-read operation	2
		SM-write operation	2
NN		packet receive	2
		packet send	8

* in machine cycles

5 Concluding points

Striking progress in computer technology has given us single-chip computers whose processing power far exceeds that of the first-generation computers. There are also various high-level languages, operating systems, and data-base systems. As a result, programs for almost any kind of application can be written, provided that their algorithms can explicitly be described. This means that computers can replace people in many areas because of their high-speed processing and large memory capability. However, there remain many application fields with hard-to-solve problems. One such is the knowledge-information processing field, where FGCS are expected to play an important role.

A machine to cope with knowledge-information processing should support extensive storage of data and high-speed inference using the data. Up to now, inference procesing has involved implementing functional and logic programming languages on conventional sequential computers. However, the need for processing power of new applications in knowledge-information processing may exceed the capabilities of sequential computers.

The architecture of parallel inference mechine makes it a possible candidate for coping with such processing requirements. Computer architectures proposed for parallel inference machines include the high-level language machine [10] as well as the data flow machine.

6 Literature

[1] P. Bishop, Fifth generation computers: concepts, implementations and uses, (Ellis Horwood, 1986).

[2] K. Furukawa, T. Yokoi, Basic software system, in: ICOT, ed., Proc. Int'l Conf. Fifth Gener. Comp. Systems 1984, (North-Holland, 1984) 37-57.

[3] L. O. Hertzberger, R. P. Van De Riet, Progress in the fifth generation inference architectures, Future Generation Computer Systems **1** (2) (1984) 93-102.

[4] N. Ito, H. Shimizu, M. Kishi, E. Kuno, K. Rokusawa, Data-flow based execution mechanisms of Parallel and Concurrent Prolog, New Generation Computing **3** (3) (1985) 15-41.

[5] N. Ito, M. Kishi, E. Kuno, K. Rokusawa, The dataflow-based parallel inference machine to support two basic languages in KL1, in: J.V. Woods, ed., Fifth Gener. Comp. Architectures, (North-Holland, 1986) 123-145.

[6] N. Ito, M. Sato, E. Kuno, K. Rokusawa, The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D, Proc. 13th Int'l Symp. Comp. Arch., (IEEE, 1986) 149-156.

[7] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata, T. Murayama, The architecture of a parallel inference engine - PIE, in: ICOT, ed., Proc. Int'l Conf. Fifth Gener. Comp. Systhems 1984, (North-Holland, 1984) 479-488.

[8] K. Murakami, T. Kakuta, R. Onai, Architectures and hardware systems: parallel inference machine and knowledge base machine, *in: ICOT, ed., Proc. Int'l Conf. Fifth Gener.* Comp. Systems 1984, (North-Holland, 1984) 18-36.

[9] K. Murakami, T. Kakuta, R. Onai, N. Ito, Research on parallel machine architecture for fifth-generation computer systems, Computer **18** (6) (1985) 76-92.

[10] H. Tanaka, A parallel inference machine, Computer **19** (5) (1986) 48-54.

PODATKOVNO PRETOKOVNI PARALELNI STROJ ZA SKLEPANJE. V članku je predstavljen paralelni stroj za sklepanje, ki temelji na podatkovno pretokovnem izvrševanju logičnih programov. Stroj podpira izvrševanje logičnih programov, zapisanih v OR ali AND-paralelnem Prologu (Parallel Prolog, PARLOG, Concurrent Prolog, GHC). Takšni programi se prevedejo v podatkovno pretokovne programske grafe, ki ustrezajo strojnemu jeziku. Podan je primer transformacije programa, zapisanega v jeziku GHC, v ustrezni podatkovno pretokovni programski graf. Arhitektura stroja obsega procesne elemente ter strukturne pomnilnike, ki jih povezuje hierarhična mreža. Procesni elementi izvršujejo dele programskega grafa sočasno, pri čemer si delijo podatke, zapisane v strukturnih pomnilnikih. Podan so tudi prostorske in časovne zahteve posameznih komponent arhitekture.