ICOT Technical Report: TR-168

TR-168

Guarded Horn Clauses and Experiences with Parallel Logic Programming

by
Jiro Tanaka, Kazunori Ueda, Toshihiko Miyazaki
Akikazu Takeuchi, Yuji Matsumoto and Koichi Furukawa

April, 1986

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Guarded Horn Clauses and Experiences with Parallel Logic Programming
(Guarded Horn Clauses とそれによる並列プログラミング)

Jiro Tanaka[*], Kazunori Ueda, Toshihiko Miyazaki, Akikazu Takeuchi,
(田中二郎　上田和紀　宮崎敏彦　竹内彰一)

Yuji Matsumoto, and Koichi Furukawa[◎]
(松本裕治　古川康一)

ICOT Research Center, Mita-Kokusai-build 21F
1-4-28,Mita, Minato-ku, Tokyo 108, Japan
Phone 456-2514　Fax 456-1618
(財団法人　新世代コンピュータ技術開発機構)
(〒108　港区　三田　1-4-28、　三田国際ビル21階)
(☎456-2514　Fax 456-1618)

[Abstract]

This paper tries to overview the various activities of ICOT related to Guarded Horn Clauses (GHC).

We describe a new parallel logic language GHC, which is proposed by Ueda [Ueda 85c, Ueda 86] first. The features of this language exists in simplicity and the ease of implementation. Then the implementation of GHC and its programming efforts are described.

We also summarize the current status of Kernel Language Version 1 (KL1). KL1 is the language system for the PIM hardware [Murakami 85a]. The overall structure of KL1 and its distributed implementation efforts are described.

1. Introduction

The final goal of the Fifth Generation Computer Project is the development of logic-based high-speed parallel-computing-system. Our objective is the design and development of a "logic programming language" which allows parallel execution. Kowalski [Kowalski 74] has pointed out that a set of Horn clauses also allows "parallel" execution as well as sequential execution. Various efforts have been carried out for the or-parallel or and-parallel execution of a Prolog program. However, our view for such efforts is that these approaches may be inadequate for parallel programming in general, although they may be useful for uncontrolled all-solution-search problems. What we want is a more expressive, general-purpose parallel programming language which includes important concepts such as processes, communication and synchronization.

2 Parallel logic programming language

2.1 Design Goals

The design requirements for our parallel logic programming language can be summarized as follows [Ueda 85c]:

-- Parallelism. It must express parallelism "by nature." Sequential languages which are added with parallel constructs are inadequate. It should keep as little sequentiality as possible in order to preserve parallelism inherent in a Horn-clause program.

-- Expressiveness. It must be an expressive, general-purpose parallel programming language. In particular, it must be able to express important concepts in parallel programming such as processes, communication, and synchronization.

-- Simplicity. We do not have much experience with parallel programming. Therefore, it must be a "simple" language and we must establish a foundation of parallel programming first.

-- Efficiency. There are various typical parallel problems to be described in the language. It is important that we can execute such programs as efficiently as the comparable ones written in other existing parallel programming languages.

2.2 GHC

The languages we are interested in are the parallel logic languages such as Parlog [Clark 85] and Concurrent Prolog [Shapiro 83]. These languages seem to be most close to the above mentioned requirements. Although there are differences, the basic computation mechanisms of these languages are quite similar: Horn clauses with guards are used for defining predicates, goals are executed in parallel, and they have some synchronization mechanisms between goals.

In this section, we describe a new parallel logic language GHC, which was proposed by Ueda [Ueda 85c, Ueda 86]. It inherits many features from Parlog and Concurrent Prolog. What is most characteristic is that the guard is the only syntactic construct added to Horn clauses. In GHC, synchronization is realized by the suspension mechanism of guards.

A GHC program is a set of guarded Horn clauses of the following form:

```
        H  :-   G1, G2, ... , Gm  |  B1, B2, ... , Bk .
```

The operator | is called a commitment operator. The part of a clause before | is called a guard, and the part after | is called a body. Note that a clause head is included in a guard.

A goal clause has the following form:

```
:- B1, ... , Bn.  (n > 0)
```

This can be regarded as a guarded Horn clause with an empty guard.

The semantics of GHC is quite simple. Informally, to execute a program is to refute a given goal clause by means of input resolution using the clauses constituting the program. This can be done in a fully parallel manner under the following rules.

(a) Unification invoked in the guard of a clause cannot instantiate the caller of that clause. In such a case, the unification suspends until that caller will be instantiated by some other goal. This provides the basic synchronization mechanisms of GHC.

(b) When the guard of a clause succeeds, it is first confirmed that no other clause has been selected for the same goal. If confirmed, that clause is selected exclusively for subsequent execution of the goal.

(c) Unification invoked in the body of a clause cannot instantiate the guard of that clause until that clause is committed.

It must be stressed that under the rules stated above, anything can be done in parallel: Conjunctive goals can be executed in parallel; candidate clauses for a goal can be tested in parallel; head unification can be done in parallel; head unification and the execution of guard goals can be done in parallel. However, it would have to be even more stressed that we can also execute a set of tasks in an arbitrary order as long as it does not change the meaning of the program.

2.3 Program examples

In this section we give program examples to show how GHC programs are described.

Binary Merge

```
merge([A|Xs], Ys,     Zs) :- true | Zs=[A|Zs1], merge(Xs, Ys, Zs1).
merge(Xs,     [A|Ys], Zs) :- true | Zs=[A|Zs1], merge(Xs, Ys, Zs1).
merge([],     Ys,     Zs) :- true | Zs=Ys.
merge(Xs,     [],     Zs) :- true | Zs=Xs.
```

The goal merge(Xs, Ys, Zs) merges two streams Xs and Ys (implemented as lists) into one stream Zs. This is an example of nondeterministic programs. Note that no binding can be exported from the guard; the binding to Zs must be done in the body.

Generating Primes

```
primes(Max, Ps) :- true | gen(2, Max, Ns), sift(Ns, Ps).
gen(N, Max, Ns) :- N <= Max | Ns=[N|Ns1], N1 := N+1, gen(N1, Max, Ns1).
gen(N, Max, Ns) :- N >  Max | Ns=[].
sift([P|Xs], Zs) :- true | Zs=[P|Zs1], filter(P, Xs, Ys), sift(Ys, Zs1).
sift([],     Zs) :- true | Zs=[].
filter(P, [X|Xs], Ys) :- X mod P=:=0 |              filter(P, Xs, Ys).
filter(P, [X|Xs], Ys) :- X mod P=\=0 | Ys=[X|Ys1], filter(P, Xs, Ys1).
filter(P, [],     Ys) :- true        | Ys=[].
```

The goal primes(Max, Ps) returns through Ps a stream of primes up to Max. The stream of primes is generated from the stream of integers by filtering out the multiples of primes. For each prime P, a filter goal filter(P, Xs, Ys) is generated which filters out the multiples of P from the stream Xs, yielding Ys.

3. Sequential implementation

Even our final target of GHC is parallel implementation, sequential implementation of GHC is still important. Our experience on Concurrent Prolog shows that the interpretive execution of such language can be very slow [Ueda 85a]. Therefore, we concentrated our efforts on implementing GHC "compilers."

GHC and Prolog have lots of similarities. Therefore, translation of the former to the latter is much simpler than direct compilation to a machine language. Therefore, the approach we took is compiling GHC programs to DEC-10 Prolog programs. We already have the DEC-10 Prolog compiler which translates a Prolog program to the machine language. It means that the GHC source program will be finally translated to the machine code.

The basic technique for compiling a parallel logic programming language to Prolog has already been shown in [Ueda 85a]. By using that technique, we have already made two GHC compilers [Furukawa 85a], one developed by Miyazaki and the other developed by Ueda. These compilers have the following features in common.

(1) They both use the similar techniques to Concurrent Prolog Compiler [Ueda 85a] which compiles a Concurrent Prolog program to DEC-10 Prolog.

(2) Both systems evaluate the guards of candidate clauses sequentially. Or-parallel execution of candidate clauses is not done. The body of a clause is evaluated only after that clause is selected.

(3) Since they are both implemented on top of Prolog, there exists interface to Prolog. The goal prolog(X) calls X as a Prolog goal.

The differences between two can be summarized as follows:

(a) Ueda's Compiler [Ueda 85a]
This compiler works on the subset of GHC where there is no user-defined goal appearing in the guard of a clause. We call this subset "Flat GHC" (FGHC) since there is no nesting guards. In this case, we do not need the run time check where variables belong to. This compiler does not distinguish "failure" and "suspension," i.e., "failure" is handled as "suspension." Therefore, the execution speed is very fast (approximately 11 K LIPS for "append" program on DEC-2065).

(b) Miyazaki's Compiler
This compiler works on the full set of GHC in the sense that we can call user-defined goal from the guard of a clause. In this case, we need run time check where variables belong to. We have realized this by using "address-comparison method." The execution speed is slower compared with Ueda's compiler. This compiler distinguishes between "failure" and "suspension."

Although compilation to Prolog provides us an offhand way to execute GHC programs, we also need more direct implementation for more realistic applications. Therefore, compilation of FGHC program to the VAX11-780 machine code has also been tried. The current status is that we have just finished up the object code mock-ups and are estimating the execution speed. The implementation work is now going on using C and VAX11-780 assembly languages.

4. GHC programming

Since we do not have much experiences on parallel programming, it seems to be very important to get experiences on parallel programming. Therefore, based on GHC, various application efforts are currently carried on. These efforts can be summarized as follows:

(1) All-solution-search transformation [Ueda 85d]

We are currently working for the development of program transformation techniques which transforms a Horn clause program into a deterministic GHC program. This technique can be viewed as a technique which provides GHC with the all-solution-search ability. It has already shown that this technique can be applicable to the non-trivial class of programs and the transformed program can be executed quite efficiently. This transformation is also important in that it exploits the AND-parallelism of GHC for parallel search along with Codish's work [Codish 85].

(2) Process fusion [Furukawa 85b]

GHC enforces the stream-oriented program where computation is expressed by processes communicating with one another. In general, each process is preferably designed to compute a relatively simple and small task. However, the resulting programs, if naively implemented, may generate too many small processes, which may cause the inefficiency because of excessive interprocess communication. Process fusion is aimed at reducing the number of processes by fusing communicating processes. And this is analogous to loop fusion in procedural language. Based on the fold/unfold method by Burstall and Darlington [Burstall 77], we are currently working for the development of the program transformation method.

(3) Parallel parser [Matsumoto 86]

The aim of the work is to develop a parsing system which is naturally implemented in parallel logic programming languages. In our framework, a grammar rule written in a DCG [Pereira 80] is compiled into a program of parallel logic programming languages such as Guarded Horn Clauses and Parlog. Words in a given sentence are defined as processes, of which each consecutive pair are connected by a stream (i.e. a shared variable). Completed subtrees are also represented as processes and partially constructed parse trees are expressed as data structure produced and put into streams by such processes. The parsing operates as the dynamic construction of such processes and data. Although the construction of the parse trees proceeds from bottom to top, we also utilize top-down prediction, which is implemented as filters. A filtering process is allocated to each input stream and filters out unnecessary elements. The system is most appropriately compared with Martin Kay's Chart Parsing, in that inactive arcs correspond to processes and active arcs are represented as data passed through streams. The important feature of our method is that the grammar rules and the dictionary are completely compiled into a program in the target parallel logic programming language and the system does not need any program which interprets the grammar and the dictionary. The derived program has neither any side-effect nor duplicate computation.

(4) Algorithmic debugging [Takeuchi 86]

Another effort has been done for developing a debugger for GHC. It has been said that debugging of parallel programs is a very hard task, compared with debugging of sequential programs. The reasons why it is difficult are that 1) conceptually several computations are executed in parallel, 2) these computations may interact with each other and 3) there are new kinds of bugs such as deadlock. Usually a program is debugged by examination of execution trace of the program. Execution trace of parallel programs is, however, messy since traces of several computation are interleaved. Even if execution trace is separated into several traces using, say windows, it is not sufficient for debugging. In general, we should distinguish between debugging and understanding of a program behavior. In the case of debugging, what is required is to find the location of the bug. Monitoring of a program behavior will help finding a bug, but it forces a programmer to understand a program behavior. It is better if a programmer could debug a program only with more abstract knowledge such as input and output specification of component modules. We have defined abstract meanings of GHC programs and are developing an algorithmic debugger for GHC. The debugger is based on the idea of algorithmic debugging of Shapiro [Shapiro 82] and reduces the number of queries based on the "divide and query" strategy. Lloyd and Takeuchi formally examine the properties of the debugger [Lloyd 86]. Current version of our algorithmic debugger only deals with body of GHC clauses for simplicity.

(5) Propositional temporal logic prover [Takahashi 86]

Temporal Logic is an extension of the first order logic including the notion of time. It deals with logical descriptions and reasoning on time. A propositional temporal logic prover based on omega-graph refutation procedure has been developed in GHC.

Omega-graph refutation is a procedure to decide the validity of a temporal formula. It negates a given formula, computes the initial node formula of the negated formula, constructs the corresponding omega-graph, and checks the existence of a loop called "omega-loop." If the loop is found, then the given formula is proved to be invalid.

In our implementation, an omega-graph is constructed incrementally from initial node formula by successive node expansions. It is represented by the network of communicating processes which spawn brother processes as the graph is expanded. New methods have also been implemented for the detection of an omega-loop where the network of processes finds loops via sending and receiving messages.

We have implemented an appropriate mechanism which checkes the attempt whether it tries to access the un-constructed part of the network and forces the process to suspend. Therefore, an omega-loop detection can be performed in parallel with omega-graph construction. It contributed extensively to reduce the execution time.

5. Kernel Language Version 1

In this section, we summarize the current status of Kernel Language Version 1 (KL1). KL1 is originally the language for the PIM hardware. The prototype developments of PIM-D (Parallel Inference Machine based on Dataflow) and PIM-R (Parallel Inference Machine based on Reduction) has been done as such PIM hardware development activities [Murakami 85a]. KL1 is expected to work as the interface between PIM hardware and the software which should be developed for the parallel high-speed logic-based system.

The first conceptual specification of KL1 was fixed on 1984. In [Furukawa 84], we have stated that KL1 should be based on Concurrent Prolog [Shapiro 83] adding set-abstraction, meta-inference and module facilities.

Based on this conceptual specification, lots of enthusiastic efforts and discussions were done for the detailed specifications of KL1 [Ueda 85b]. Several implementation efforts have also been done for Concurrent Prolog [Miyazaki 85]. These efforts made it explicit problems existing in Concurrent Prolog regarding the semantics and parallel execution [Ueda 84]. It forced us to revise the conceptual specification of KL1.

After the elaborate examination, we have decided to adopt the Flat GHC (FGHC) as the core part of KL1. The discussion for the detailed specification of KL1 also has made clear that we should separate the language into several layers. Therefore, now the KL1 is the language system which consists of three layers. These are shown in Figure 1.

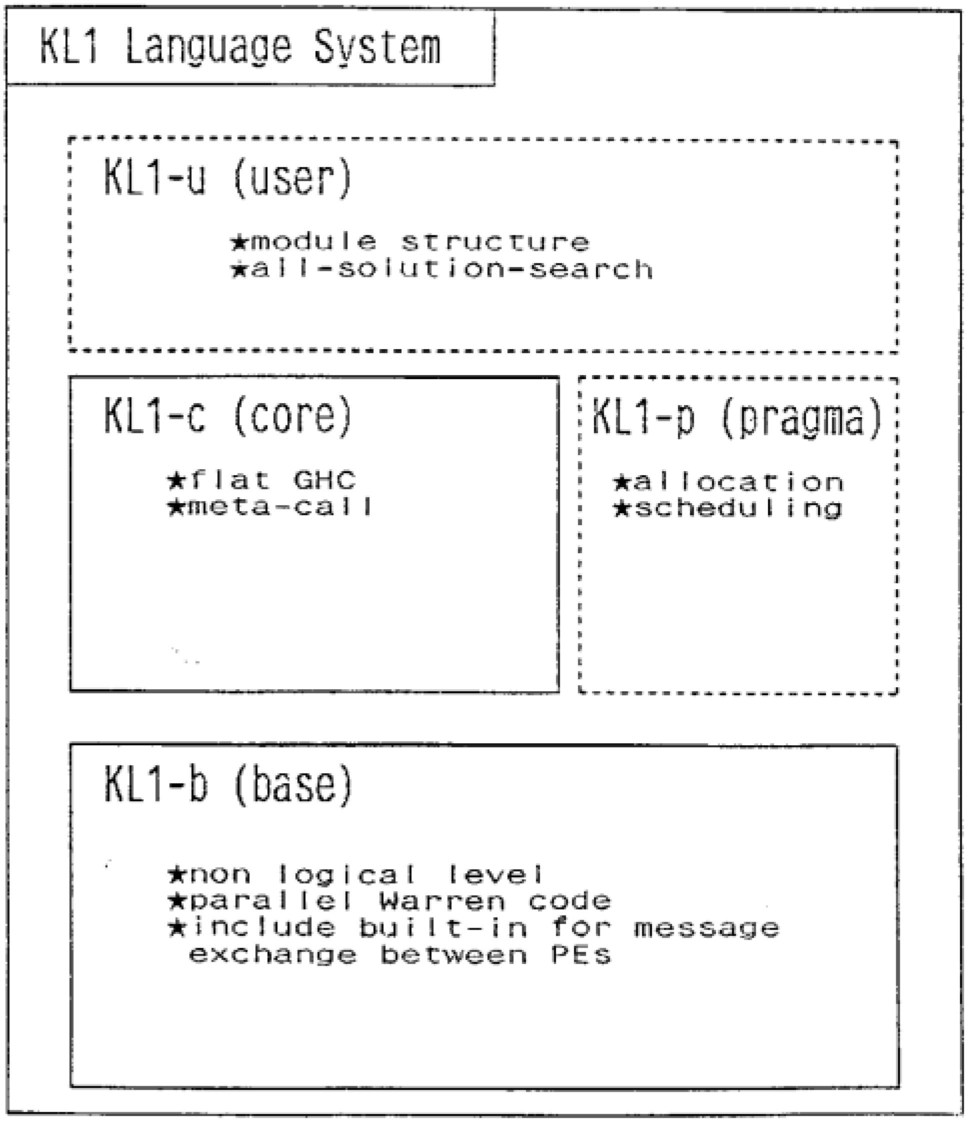

Figure 1 KL1 Language system

Here, KL1-c is the core language of KL1. We assume FGHC which includes meta-call predicates as KL1-c.

KL1-p is the pragma language which specifies how the program should be executed in a distributed/parallel environment. KL1-p is not a language as it stands. It is attached to KL1-c to specify process allocation and scheduling information.

KL1-u is the user language which should be positioned on KL1-c and

KL1-p. It includes the module structure and all-solution-search predicates. Currently, KL1-u is in the designing stage.

KL1-b is the machine language which hardware/firmware will directly support. This level is not a logical level and it looks like the parallel version of Warren's abstract machine instruction set [Warren 83].

6. Distributed implementation

Various researches have been carried out for the development of PIM machine from the "architectural" view point. However, in accordance with the development of our research, we noticed that there exists lots of problems to be solved at "software" or "firmware" level, such as (1) how to boot the system, (2) how to deliver object program to each PE, (3) how to handle input/output and interrupt, (4) how to balance the load between PEs, etc.

These problems are not the hardware problem in itself. Therefore, ICOT decided to start "Multi-PSI" project to examine such "software" problems in distributed environment. There is not so much new in hardware. ICOT has already developed Personal Sequential Inference (PSI) machine [Taki 84]. Multi-PSI hardware is simply built up by connecting 6 - 16 Personal Sequential Inference (PSI) machines with high speed grid-hardware.

6.1 The multi-PE model

The model we assumed is the multi-PE system where dozens of PEs are grid-network connected. There exist lots of designing choices. Lots of intensive discussions had been done inside ICOT. The decisions we have made are as follows:

(1) PE must be connected in a way which allows the increase of PE number. Our system must work even if we have hundreds of PE.

(2) Each Processing Element (PE) has its local memory. It has no shared memory nor global address space. In our system, PE communicates each other only via message exchange.

(3) We assume and-parallel logic language as our underlying logic language. And-parallel execution of a program is also assumed. Our claim is that and-parallelism plays more basic roles than or-parallelism in distributed environment. Each PE executes a program independently or cooperatively.

Figure 2 shows how the multi-PE model looks like.

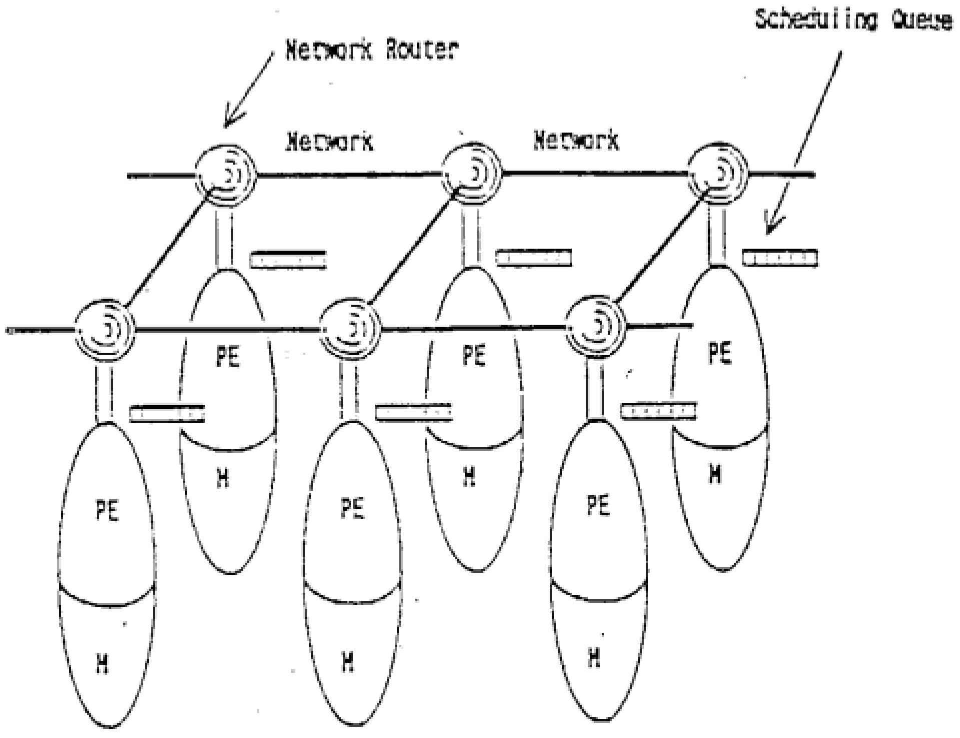

Figure 2 Multi-PE Model

This multi-PE model has the following features.

(1) We assume that the program, i.e., the definition of clauses, has been loaded to every PE from the beginning for simplicity. More realistic assumption such as "lazy" fetching of program code may be needed in a later stage.

(2) First, a goal is put into one PE, this automatically starts the computation. Each PE executes the goals which are thrown from other PEs besides processing local goals. When there is no goals to be processed on all PEs, it means the end of computation.

(3) Each PE has one scheduling queue. Each PE repeats to dequeue a goal from the scheduling queue and reduces it to the resulting goals. These goals are enqueued to the scheduling queue or thrown to other PE.

(4) Since each PE has independent address space, the unification of two variables existing on different PEs invokes the necessity to span inter-PE reference chains. Each PE has the variable management table for that purpose.

(5) Unification sometimes invokes various messages to other PEs. The examples of such messages are "get_value," "unify," "unify_channels," etc.

6.2 Multi-PE simulator

We have implemented the software simulator of the multi-PE system [Tanaka 86]. In relate to this simulator, our system is based on the work by Miyazaki and Murakami [Murakami 85b]. Our system is written in Prolog and simulates the execution of pre-processed FGHC program in a multi-PE environment. Our system consists of processes and a network manager as is shown in Figure 3.

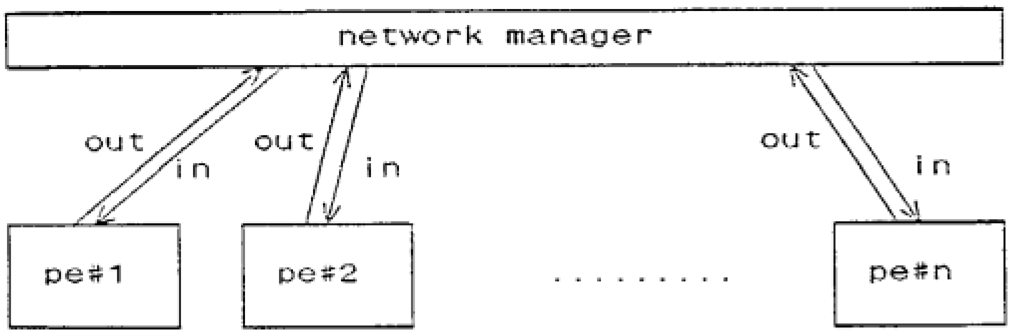

Figure 3 Multi-PE simulator

Each PE checks whether there is message from its input channel. If so, messages are added to the scheduling queue. Then it executes the first goals inside the queue. If there is a message to be sent, the message is set to the output channel. Network manager takes care of message exchange between PEs. Messages inside output channels include the sending address. Network manager delivers the message to the specified input channel.

Since our system does not have global address space, sending of a goal which includes variables to other PE needs a little complicated mechanism. In our implementation, each PE has a variable management table. We consult this table whenever the need for inter-PE reference is generated.

6.3 Implementation on PSI [Miyazaki 86]

Implementation on PSI is more realistic effort for "Multi-PSI" project. PSI machine is a personal workstation and its designing concept is very similar to LISP machine. Conventional techniques have been adopted and ESP [Chikayama 84], which is the object-oriented extension of Prolog, is firmware supported.

The compiler which translates a FGHC program to Warren-like abstract machine instruction sequences [Warren 83] and the emulator of that abstract machine instruction set have already implemented. Our Warren-like instruction set also includes the primitives for distributed execution. Emulator has been written in ESP and it has been realized using heap are on PSI machine.

The current version only executes a program on one PSI machine. The execution speed is approximately 0.9 K LIPS on naive reverse program [Miyazaki 86]. Currently, the enthusiastic efforts for Multi-PSI system are in progress. ICOT has already finished up the design of connecting hardware. The actual hardware of Multi-PSI Version 1 will be completed by the end of April 1986. The effort which extends this implementation to Multi-PSI system is also going on.

7 Summary

We described the various activities of ICOT related to Guarded Horn Clauses (GHC). The language features of GHC, its implementation efforts and its applications were described. We also summarized the current status of KL1 and its distributed implementation efforts. These activities are on-going project and must be extended further on various directions.

8. Acknowledgments

This research was carried out as a part of the Fifth Generation Computer Project. We would like to thank our colleagues, Kazuo Taki and Takashi Chikayama, for their useful comments and suggestions. We also would like to thank Kazuko Takahashi and Tadashi Kanamori at Central Research Laboratory, Mitsubishi Electric Corp. for their GHC programming efforts. Thanks also goes to Kazuhiro Fuchi, the Director of ICOT for giving us the opportunity to pursue this research.

[References]

[Burstall 77] Burstall, R. and Darlington, J.: A Transformation System for Developing Recursive Programs, JACM Vol.24, No.1, pp.44-67.

[Chikayama 84] Chikayama, T. : ESP Reference Manual. ICOT Technical Report TR-044, ICOT, 1984.

[Clark 85] Clark, K., Gregory, S.: PARLOG: Parallel Programming in Logic. Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, Revised June 1985.

[Codish 85] Codish, M.: Compiling OR-parallelism into AND-paralelism, M.Sc Thesis, Weizmann Institute of Science, December 1985.

[Furukawa 84] Furukawa K. et al.: The Conceptual Specification of the Kernel Language Version 1. Technical Report TR-054, ICOT, 1984.

[Furukawa 85a] Furukawa K. et al.: Kernel Language Version 1, explanation materials, ICOT, 1985, in Japanese.

[Furukawa 85b] Furukawa K., Ueda K.: GHC Process Fusion by Program Transformation. In Proc. Second National Conf. of Japan Society of Software Science and Technology, 1985, pp. 89-92.

[Kowalski 74] Kowalski, R.: Predicate Logic as Programming Language. In Proc. IFIP '74, North-Holland, Amsterdam, London, 1974, pp.569-574.

[Lloyd 86] Lloyd, J. and Takeuchi, A.: A Framework for Debugging GHC. To appear ICOT TR, 1986.

[Matsumoto 86] Matsumoto, Y.: A Parallel Parsing System for Natural Language Analysis, to be presented at the Third Int. Logic Programming Conf., London, 1986.

[Miyazaki 85] Miyazaki, T. et al.: A Sequential Implementation of Concurrent Prolog Based on Shallow Binding Scheme. Proceedings of 1985 Symposium on Logic Programming pp. 110-118.

[Miyazaki 86] Miyazaki, T. and Taki, K.: The implementation method of Flat GHC on Multi-PSI system. Unpublished draft, 1986, in Japanese.

[Murakami 85a] Murakami, Kunio et al.: Research on Parallel Machine Architecture for F.G.C.S.. Computer, vol.18, No.6, June 1985.

[Murakami 85b] Murakami, Kenichiro: The study of unifier implementation in multi-processor environment. Multi-SIM study group internal document, ICOT, 1985, in Japanese.

[Pereira 80] Pereira, F.C.N. and Warren, D.H.D.: Definite Clause Grammars for Language Analysis - A survey of the Formalism and a Comparison with Augmented Transition Networks, Artificial Intelligence, 13, pp.231-278, 1980.

[Shapiro 82] Shapiro, E.: Algorithmic Program Debugging, MIT Press, Cambridge, Mass, 1982.

[Shapiro 83] Shapiro, E.: A Subset of Concurrent Prolog and its Interpreter. ICOT Technical Report TR-003, ICOT, 1983.

[Shapiro 85] Shapiro, E. et al: Logix User Manual for Release 1.1. Weizmann Institute, Israel, 1985.

[Takahashi 86] Takahashi,K. and Kanamori,T.: On Parallel Programming Methodology in GHC. ICOT Technical Report, to appear.

[Takeuchi 86] Takeuchi, A.: Algorithmic debugging of GHC programs and its implementation in GHC, unpublished draft, 1986.

[Taki 84] Taki, K. et al.: Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT, pp.398-409.

[Tanaka 86] Tanaka, J. et al.: Distributed Implementation of FGHC -- Toward the realization of Multi-PSI system --. Unpublished draft, 1986.

[Ueda 85a] Ueda, K. and Chikayama, T.: Concurrent Prolog Compiler on Top of Prolog. Proc. of 1985 Symposium on Logic Programming, pp.119-126, 1985.

[Ueda 85b] Ueda, K.: Concurrent Prolog Re-Examined. ICOT Technical Report TR-102, ICOT, 1985.

[Ueda 85c] Ueda, K.: Guarded Horn Clauses. ICOT Technical Report TR-103, ICOT, 1985.

[Ueda 85d] Ueda, K.: Making Exhaustive Search Program Deterministic. ICOT Technical Report TR-145, ICOT, 1985. Also to be presented at the Third Int. Logic Programming Conf., London (1986).

[Ueda 86] Ueda, K.: Guarded Horn Clauses. Doctoral Thesis, Information Engineering Course, Faculty of Engineering, University of Tokyo, March 1986.

[Warren 83] Warren, D. H.: An Abstract Prolog Instruction Set. Tech. Report 309, Artificial Intelligence Center, SRI International, CA, 1983.