TR-232

An Evaluation of the FGHC via Practical Application Programs

by

M. Kishimoto, A. Hosoi,

K. Kumon and A. Hattori

FUJITSU Ltd.

February, 1987

ICOT

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

(03) 456-3191～5
Telex ICOT J32964

An Evaluation of the FGHC via Practical Application Programs*

Mitsuhiro Kishimoto, Akira Hosoi,
Kouichi Kumon, Akira Hattori

FUJITSU LIMITED

1015, KAMIKODANAKA, NAKAHARA-KU,
KAWASAKI 211, JAPAN
kiss@flab.flab.fujitsu.junet
hosoi@flab.flab.fujitsu.junet
kumon@flab.flab.fujitsu.junet

Abstract FGHC (Flat Guarded Horn Clauses) is a family of parallel logic programming languages. It has been chosen as the kernel language for the Parallel Inference Machine (PIM) of the Japanese fifth generation computer project (FGCS). To offer valuable suggestions for the decisions on design of a high performance PIM, we implemented an efficient language processor for FGHC and developed practical parallel application programs written in FGHC. These application programs behaved differently from other well-known small example programs. We have also refined our methods for implementation of the language processor according to measurement and analysis and discussed the required functions of the PIM.

This research was sponsored by MITI as a part of the Japanese FGCS project.

KEYWORDS: LOGIC PROGRAMMING, PARALLEL PROCESSING, AND-PARALLELISM, FGHC, PIM, CONTEXT SWITCH, SUSPENSION.

*To be submitted at the Fourth Symposium on Logic Programming, San Fransisco, August 1987

1. Introduction

We are developing a new high performance Parallel Inference Machine (PIM) as a part of the Japanese fifth generation computer project (FGCS)[1]. FGHC (Flat Guarded Horn Clauses [2]), a family of parallel logic programming languages [3, 4], has been chosen as the kernel language for the PIM.

Precise analysis of practical parallel application programs is essential for designing a high performance PIM. Until recently only small example programs have been used for this purpose. But the behavior of these programs seems to differ from the behavior of the typical knowledge processing programs that will run on the PIMs in the future.

We began our research on a high performance PIM through measurement and analysis of practical parallel application programs. For this purpose, we first implemented an efficient FGHC language processor consisting of an interpreter and an emulator. Meanwhile, we developed practical parallel application programs in FGHC. We picked up a parallel switchbox wire router, a VLSI CAD program, for an in-depth analysis.

We measured many dynamic characteristics of these practical programs. They behaved differently from the small example programs. Based on an analysis of the measured data, we refined our implementation methods for the language processor suitable for the practical programs. These are reflected in the design of our PIM.

We will describe the sequential implementation of FGHC in Section 2. Section 3 is an explanation of the practical parallel application program, the parallel switchbox wire router. In Section 4, we show many of the dynamic characteristics of the parallel wire router and verify the appropriateness of the implementation techniques. We will discuss the refinement of the language processor and the required characteristics for the PIM.

2. Sequential Implementation of FGHC

2.1 Language specification

FGHC is similar to standard Prolog [5]. Symbols that begin with an uppercase letter are variables and ones that begin with a lowercase letter are function symbols, predicate symbols or atoms. A program in FGHC is a collection of guarded Horn clauses. A guarded Horn clause is a logical implication of the following form with guard goals

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

where H, G1, ... Gm, B1, ... Bn are atomic formula. H is called the clause head, the Gi's are called guard goals, and the Bi's are called body goals. Trust operator " | " separates the passive part (Gi's) from the active part (H and Bi's). Logically, the guarded Horn clause should be read as "H is implied by the body B1, ... , Bn if the guards G1, ... Gm terminates successfully". A set of all clauses whose heads have the same predicate symbol with the same arity is called a procedure. A procedure is the logical disjunction of its Horn clauses.

A goal is the basic execution unit of FGHC. The role of the FGHC's goals are very similar to processes in operating systems. So, they are sometimes called a process. A goal clause starts the execution. The goal clause has the following form.

:- B1, ... , Bn.

This can be regarded as a guarded Horn clause without a passive part. Execution of FGHC is regarded as the reduction of goals under a procedural interpretation. Goal reduction tries to unify the currently executing goal and the head of a clause, and to execute all the clause's guard goals. If the head is unifiable and all guard goals terminate successfully, the body goals are spawned. Even if there is more than one clause that satisfies the above conditions, only one clause can be committed nondeterminately. Unlike

Prolog, FGHC is neither able to backtrack nor to try all clauses. If one of the clauses is committed, the others are abandoned and never considered again.

Unification in FGHC is extended to include communication and synchronization between goals. In the passive part of a clause, variables in the caller goal can be read, but attempting to instantiate the variable suspends the goal. This suspended goal is resumed when another goal instantiates the variable, on which the suspended goal is hooked. On the other hand, there are no restrictions on the unification of the active part of the clause.

2.2 Language Processor

To examine practical parallel application programs written in FGHC, we developed a sequential implementation of the FGHC language processor on Fujitsu's S3500 super mini computer. The language processor must provide a debugging aid for FGHC programs. Also, it must run at least as fast as an efficient implementation of Prolog. Meeting these demands, our language processor consists of an interpreter and an emulator. In this paper, the discussion is based mainly on the FGHC interpreter.

(1) Interpreter

Parallel programming is a relatively unknown field, so programmers in this field need powerful debugging aids. Our interpreter is well suited to meet this need.

Even in the interpreter, execution speed is still an important factor, because the debugging period will be long and the programmer needs high speed processing to achieve fast turn around. The interpreter must run almost the same speed as efficient Prolog interpreter. Thus the interpreter was written in C, and we adopted efficient methods and strategies described in the next section.

(2) Emulator

Warren's abstract instruction set [6] is well-known as a high performance Prolog abstract architecture, which we call WAM for short. The FGHC emulator adopts an abstract instruction set KL1-B (kernel language one base [7]). KL1-B is a refinement of WAM that is tuned for FGHC.

The emulator is also written in C. Its main design criteria is speed. On the emulator, the example programs run almost 10 times faster than on the FGHC interpreter.

Although, the interpreter and the emulator have the same data structures and the same execution model, but they have not yet been integrated. This work will be done in the future.

(3) Compiler

An FGHC compiler generates efficient KL1-B codes to run on the emulator and was written in Prolog.

2.3 Sequential Implementation

We adopted the following implementation methods and strategies for our language processor. Based on the behavior of practical programs we will discuss their appropriateness and will refine them in Section 4.

(1) Non-busy-waiting strategy (NBWS)

The execution speed of a language processor that adopts a busy-waiting strategy, decreases in proportion to the number of suspended goals. Because practical FGHC programs are expected to have huge number of suspended goals, a non-busy-waiting strategy should be more efficient.

(2) Multiple waiting on unbound variables

A suspended goal, which is multiple waiting on some uninstantiate variables, can be resumed when one of the variable is instantiated. There

are two methods for implementing the multiple waiting. Both methods hook the suspended goal to the unbound variables that caused the suspension. In the first method, when the goal is resumed, the resumed goal kills all the other hooks at that time. In the second method a resume flag that is shared by all hooks is turned off when the goal is resumed. According to the resume flag, the other hooks kill themselves when their variables are instantiated.

The second method omits the cost of searching and killing the other hooks. We adopted this method for multiple waiting on unbound variables.

(3) Storage of the goal record

Context switches, the exchange of the currently executing goal, happen very frequently in FGHC execution. Warren's implementation, in which the goal record is kept in the CPU, cannot increase its efficiency for FGHC, because the cost of writing the context switched goal back into main memory is greater than access advantage of keeping the goal record in the CPU.

To avoid this, we only load a pointer to the current goal record into the CPU instead of the whole goal record. This method is suitable for an execution model that switches context frequently.

(4) Bounded depth-first strategy

We adopt the bounded depth-first strategy for our goal execution. The depth-first strategy decreases the number of context switches. It also decreases the number of existing goal records.

Bounded means that the reduction has a depth limit. The bound forces a context switch at least every 100 reductions. This keeps the execution fair. If we solve the generator-and-consumer problem without any bound, generator's goal would run forever and the consumer's goal would never execute.

3. Application programs

We expected that the dynamic characteristics of small example programs (see *Figure 1*) would be different from the characteristics of typical knowledge processing programs, which will run on the PIMs in the future. The analysis of the small example program might mislead us.

We must discuss the refinement of the language processor and design of PIM hardware based on the analysis of the practical programs not to make such misstep. Thus we developed a practical parallel program written in FGHC and measured its characteristics. The program must satisfy the following conditions to be useful for this purpose.

```
append([H|T], Y, Z):- Z = [H|Z1], append(T, Y, Z1) .
append([], Y, Z):-     Z = Y .

nrev([], Y):- Y = [].
nrev([A|B], C):- nrev(B, D), append(D, [A], C).
```

(a) nreverse

```
primes(Max, Ps):- gen(2, Max, Ns), sift(Ns, Ps) .

gen(N0, Max, Ns0):- N0 <= Max | Ns0 = [N0|Ns1], N1:=N0+1,
                    gen(N1, Max, Ns1).
gen(N0, Max, Ns0):- N0 > Max | Ns0 =[] .

sift([P|Xs1], Zs0):- Zs0 = [P|Zs1], filter(P, Xs1, Ys), sift(Ys, Zs1).
sift([], Zs0):- Zs0 = [].

filter(P, [X|Xs1], Ys0):- A:=X mod P,A =\= 0 | Ys0 = [X|Ys1],
                          filter(P, Xs1, Ys1) .
filter(P, [X|Xs1], Ys0):- A:=X mod P, A =:= 0 | filter(P, Xs1, Ys0).
filter(P, [], Ys0):- Ys0 = [] .
```

(b) primes

```
qsort([X|Xs], Ys0, Ys3):- part(Xs, X, S, L) , qsort(S, Ys0, [X|Ys2]) ,
                          qsort(L, Ys2, Ys3).
qsort([], Ys0, Ys1):- Ys0 = Ys1 .

part([X|Xs], A, S, L0):- A < X | L0 = [X|L1] , part(Xs, A, S, L1) .
part([X|Xs], A, S0, L):- A >= X | S0 = [X|S1] , part(Xs, A, S1, L) .
part([], _, S, L):-   S = [], L = [] .
```

(c) qsort

Figure 1. small example programs

(1) Parallelism

The problem that the program will solve must naturally include a large degree of parallelism. It is inappropriate to drag out parallelism by force. The scale of parallelism needed is 10 or 100 times more than the number of processor elements in the PIM. This means the problem must naturally include from 100 to 1,000 scale parallelism.

(2) Communication and synchronization mechanism

FGHC is a parallel logic programming language based on the communication and synchronization mechanism. The small example programs, however, are based on the call and return mechanism like sequential programming languages. To evaluate FGHC in the parallel aspect, the application programs must act on the communication and synchronization mechanism.

(3) Practical problems

We developed application programs under the restriction that the programs solve a practical problem. To make the analysis relevant, we avoided constructed problems. Also the size of the application program must be more than 1,000 steps.

Futhermore, we try to show that FGHC is useful by solving practical problems in FGHC, since FGHC is not recognized as a practical programming language yet.

(4) Comparison with other languages

It is necessarily to compare with another programming language to show FGHC to be useful. The refinement of PIM hardware and language processor implementation techniques are only relative comparisons on FGHC. Another kind of comparison, with a program written in another programming language, for example C, is also important.

Unfortunately, none of the small programs shown in *Figure 1*, satisfies the above conditions. In terms of our criteria, these programs are unsuitable for the base of the design and for refinement.

We could inspect FGHC's functions by developing practical parallel programs. Also we could gain various experience in debugging parallel programs. However, these topics will be discussed in another paper.

3.1 The parallel router

We chose a switchbox wire router as our practical parallel application program [8], since it satisfies all of the above requirements. The switchbox wire router is a gate-array VLSI CAD program. It is used in the final stage of LSI design. It is characterized as follows.

(a) The switchbox is a rectangular routing area without obstructions.

(b) The signal terminals are only on the sides of the area.

(c) The terminals are on all four sides and their positions are fixed.

(d) There are two routing layers, and each terminal is on one layer or the other.

(e) Complete connection of all terminals are needed.

(f) A connection may be a non-minimum spanning path.

Figure 2 is an example of switchbox wire router problem. The numbers written on the four sides are net identifiers. The router program connects all terminals that have the same identifier. A set of these terminals is called a net. There are two routing layers, an upper and a lower one. To keep the routing algorithm simple, the upper layer has only horizontal lines, and the lower layer has only vertical ones. Vias connect the two layers.

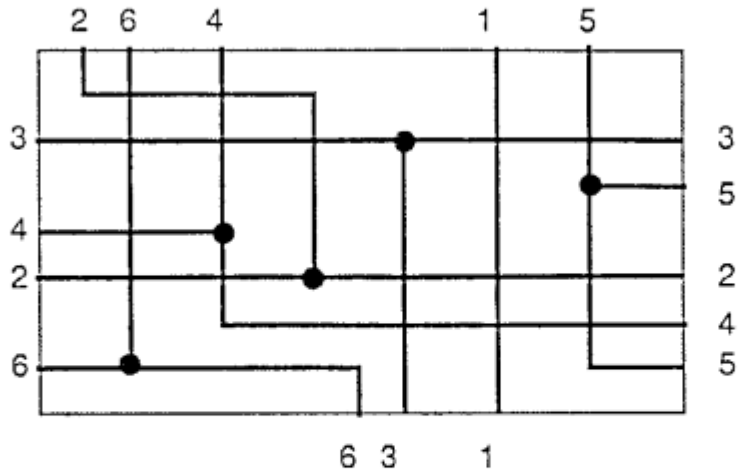


Figure 2. Switchbox wire router problem example

The flow chart of the whole parallel wire router is shown in *Figure 3*. The parallel router iterates the following operations. The router searches minimum spanning path for each net, and detects bottleneck grids on minimum spanning paths. There are indispensable grid points as a bottleneck. And the router examines whether the bottleneck grid on the minimum spanning path is a real bottleneck on the whole routing area. The operation for each net is independent from the other. The parallel router has two kinds of parallelism. Searching for the bottleneck grids simultaneously is the large grain parallelism and fine grain parallelism will be explained later. After these parallel operations, the nets try to acquire their bottleneck grids. This acquisition is sequential operation.

If all the nets have connected, the router terminates successfully. Otherwise, there are still one or more unconnected nets when all nets have acquired their bottleneck grids (they are balanced). The router tries to select and allocate additional non-bottleneck grids to one of the unconnected nets. This selection is registered for future backtracking. And the router begins to search for bottleneck grids again. The last case is a confliction of the bottleneck grids. If confliction occurs, it backtracks to the last candidate grid

selection, undoes all grids allocated after the last selection, and selects another candidate grid.

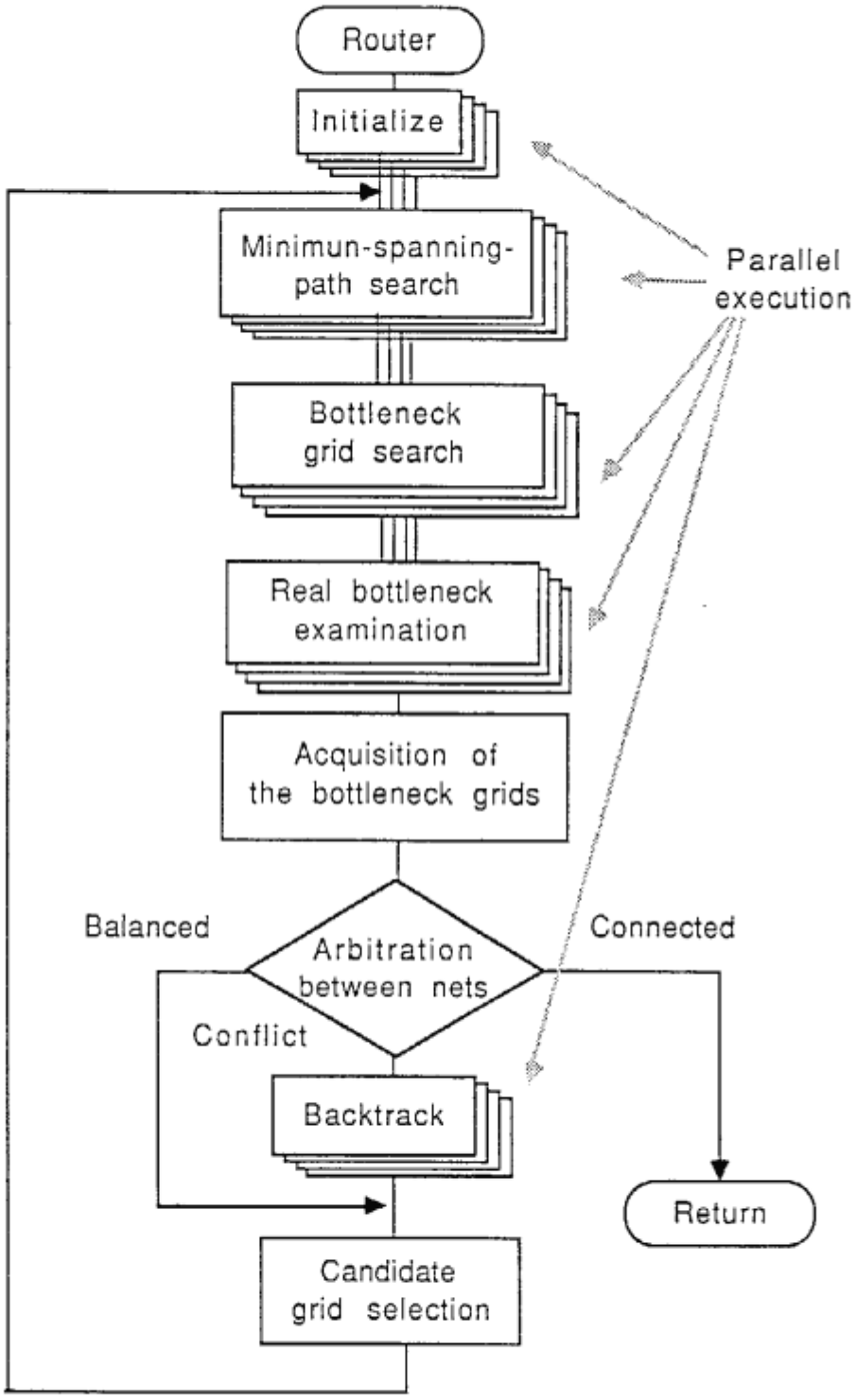


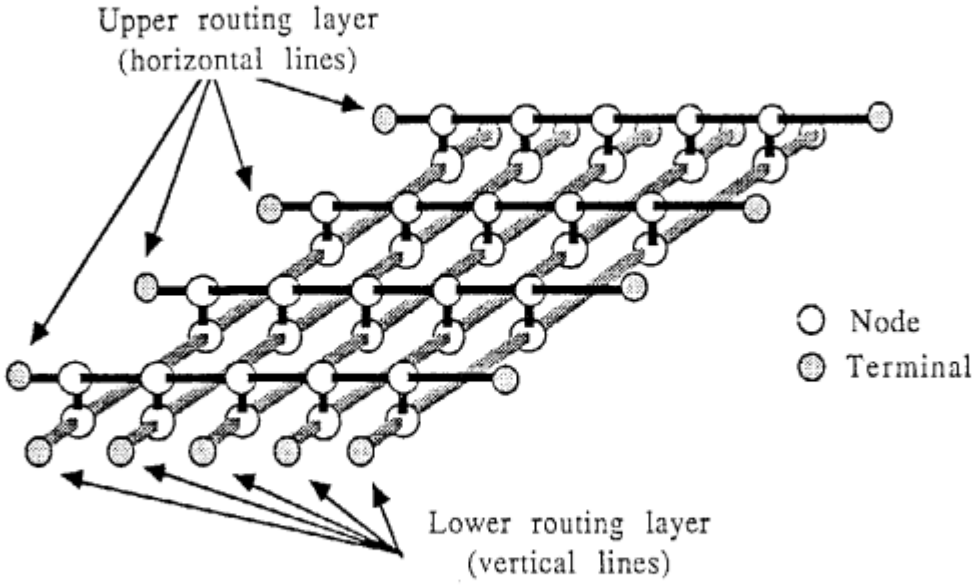
Figure 3. Wire router flow chart

The main and the most time consuming part in this router is the minimum spanning path search. We did not use the whole wire router program for our measurement, but instead used only this part. We will describe the minimum spanning path search in detail.

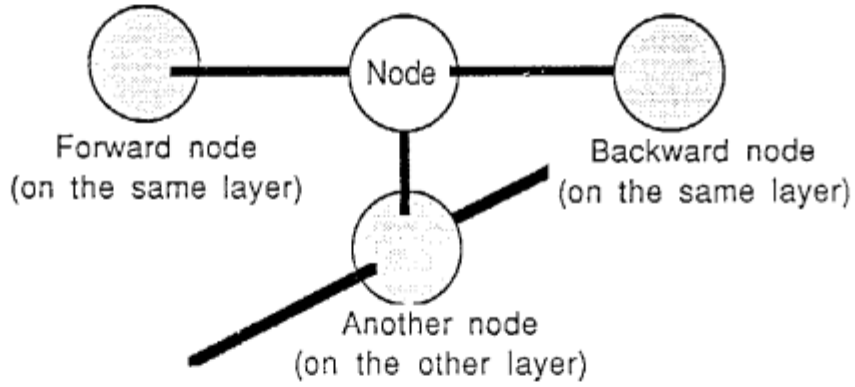
A parallel switchbox wire router, written in C has been reported [9]. Their program is run on cellular array processor (CAP). But a comparison between FGHC program and C program is beyond the limits of this paper.

3.2 Minimum spanning path search

In the wire router, many perpetual processes called nodes are placed at every grid on both layers. For example, the structure of the whole routing area (4 X 5 grids) is shown in *Figure 4* (a). *Figure 4* (b) shows the detail of each node. The minimum spanning path search is processed in terms of sending messages between nodes, and updating the states of the nodes. Many perpetual processes cooperate in parallel, since the switchbox wire router naturally has a large degree of fine grain parallelism. This is the second kind of parallelism.



(a) Structure of the routing area



(b) Structure of each node

Figure 4. Structure of a node connection

We used the following three algorithms to search for a minimum spanning path. The first and second are known as standard grid expansion algorithms.

(1) Sequential wave front (SWF) method

The SWF method is a well-known Lee-Moore type router. In terms of a labeling message, each node is labeled by its Manhattan distance from the starting terminal. All nodes that have the same label are linked. They are called a wave front. The SWF method is shown in *Figure 5*. In this figure

and also in *Figures 6* and *7*, we use one routing layer to keep the explanation simple.

The next wave front spreads sequentially from one side of the last wave front. In figure 5, we show the order of the wave front expansion. When the expansion has done, the new wave front checks whether it has reached the end terminal. If it has not, the new wave front starts a next wave front expansion. We feel there is not enough parallelism in this sequential method.

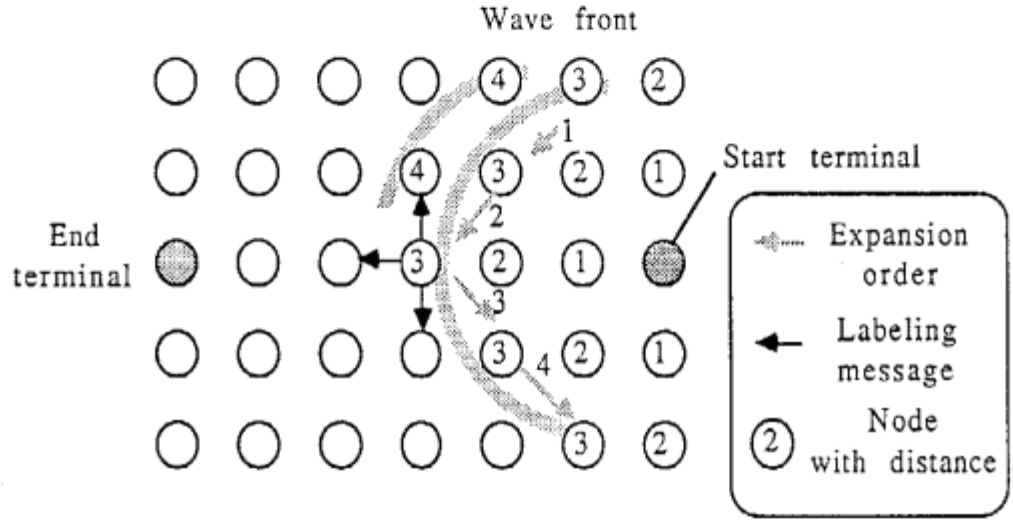


Figure 5. Sequential wave front method

(2) Volley wave front (VWF) method

This method is a refinement of the SWF method. All nodes on the last wave front watch (are suspended on) the same variable. This variable is instantiated by the check process, when the check for reaching an end terminal fails. All the suspended nodes will be resumed and start to send the next labeling messages at once.

The VWF method seems to have a large degree of parallelism. The mechanism for this method is shown in *Figure 6*.

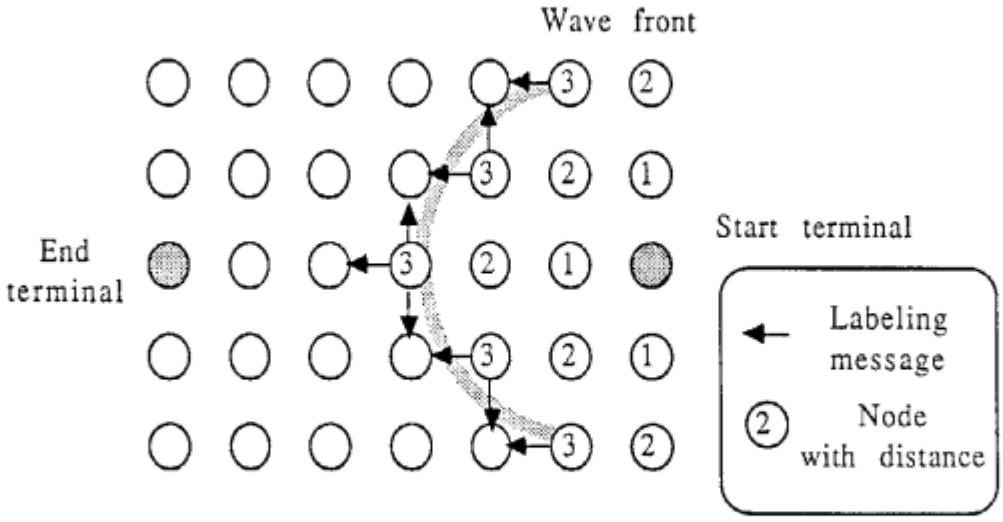


Figure 6. Volley wave front method

(3) Reckless running (RLR) method

The RLR method has little sequentiality. The mechanism for the RLR method is shown in *Figure 7*. When a node receives a labeling message with distance information, Dm, from one of the neighboring nodes, the node compares its own state Dn (current known distance) and Dm. If Dm ≥ Dn, the node ignores this labeling message. If not (Dm < Dn), it substitutes Dm for its own state and sends the new labeling message with Dm + 1 to the rest of its neighboring nodes.

This method has a huge degree of parallelism and the processing is very simple. In the worst case, however, it does a lot of useless operations.

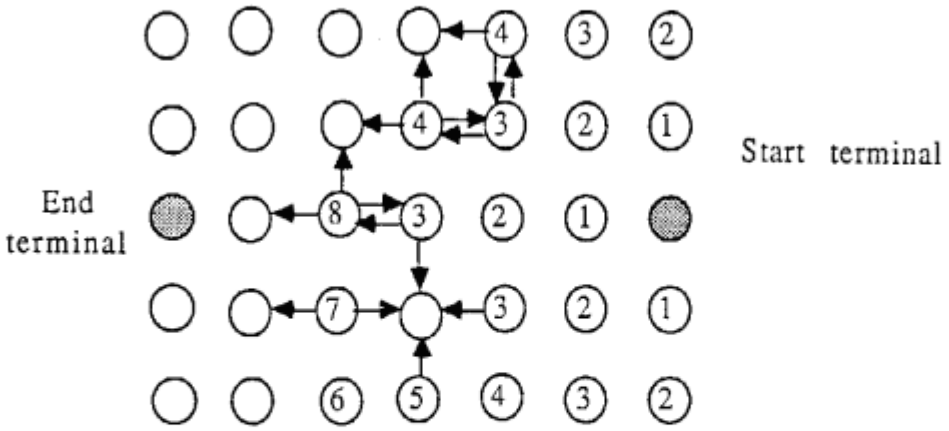


Figure 7. Reckless running method

4. Dynamic characteristics of application programs

In this section, we will explain the dynamic characteristics of the parallel wire router. Also we will discuss the appropriateness of the implementation techniques for the language processor described in Section 2, and refine them based on the analysis. We also list the functions required of the PIM hardware.

4.1 Performance

We have made two kinds of comparisons to evaluate the performance of the FGHC interpreter. The first comparison is with a Prolog interpreter written in C, because recently the Prolog interpreter has enough performance, and is used practically. The execution time for several small example programs run on both interpreters is shown in *Figure 8*. We can do these measurements only on small size programs, because it is very difficult to make two programs using the same algorithm but written in different programming languages. *Figure 8* shows the FGHC interpreter has from 42 to 80% of the performance of the efficient Prolog interpreter.

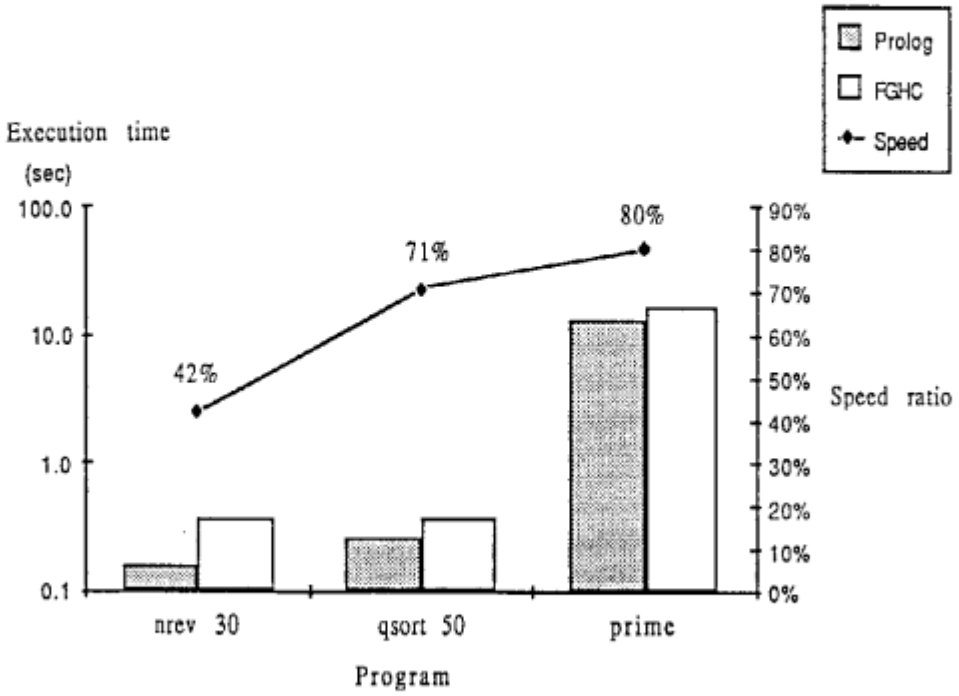
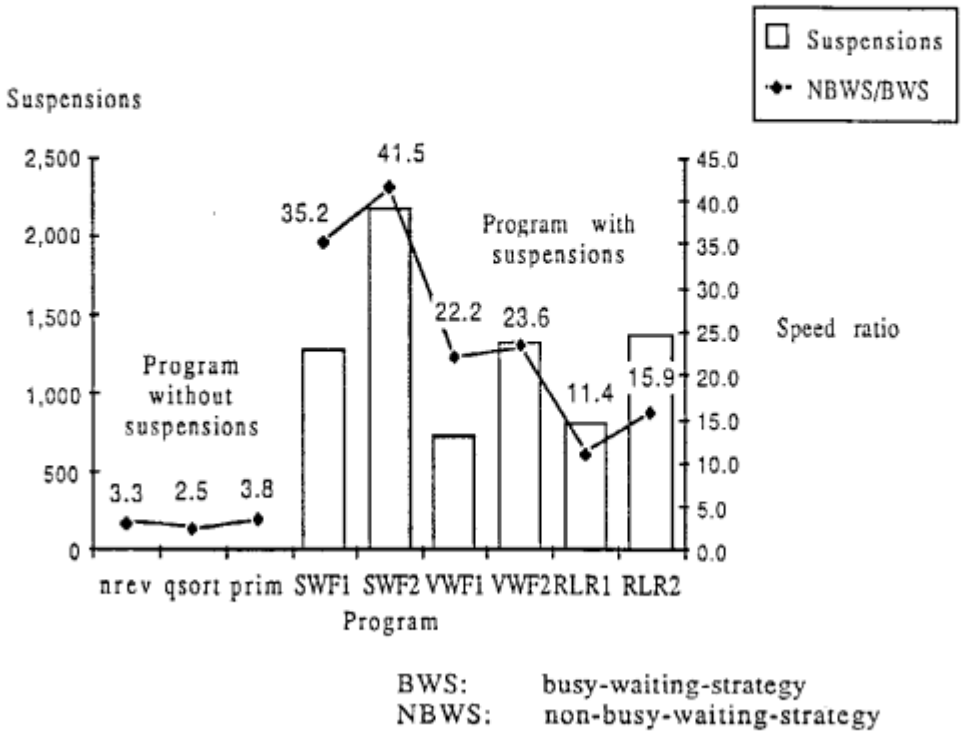


Figure 8. Comparison with a Prolog interpreter

Omitting the backtracking and related operations helps the FGHC interpreter to run fast. However, the FGHC interpreter could not use an efficient stack mechanism since the order of the goals is non-deterministic in FGHC. Furthermore, it must include a goal suspension mechanism. In the programs used for *Figure 8*, no suspension has occurred, though the overhead of the suspension checking decreases the FGHC interpreter's performance.

The second comparison verifies the advantage of the non-busy-waiting strategy (NBWS). We compared our interpreter with an FGHC compiler written by Ueda [10]. The FGHC compiler compiles from FGHC source program to Prolog object program. The compiled code uses a busy-waiting strategy (BWS). *Figure 9* shows an interesting result of the comparison. When there are none or a few suspended goals (nrev, qsort and prime in *Figure 9*), the NBWS interpreter runs only 3 times faster than BWS compiled code. Thus we could say that the busy-waiting strategy is efficient for programs without suspensions.



BWS: busy-waiting-strategy
NBWS: non-busy-waiting-strategy

Figure 9. Comparison between BWS and NBWS

When there are many suspended goals, however, the NBWS interpreter runs from 10 to 40 times faster than the BWS compiled codes. In Figure 9, SWF, VWF and RLR are in this case. The SWF, VWF and RLR indicate the method described in Section 3.2, and 1 and 2 are the program number. Practical application programs, like a wire router, would have many suspended goals. Based on this data , The NBWS is essential and the interpreter has enough performance for practical use.

4.2 Context switch

Goal reduction usually spawns some children goals. The control transfers from the reduced goal to the eldest child goal. This transformation is not a context switch. Context switch is an exchange of the currently executing goal to the non-child goal that is in the ready-goal queue.

Context switch occurs in following cases. The current executing goal reduced with a fact, clause without children goals. Or the executing goal is a built-in predicate, or it suspended on some uninstantiated variables. We show the number of reductions and the number of context switches for several programs in *Figure 10*. Roughly speaking, a context switch occurs every 2 reductions. Strictly speaking, from 2.18 to 2.74.

This ratio is lower than we would expect at first. The depth first strategy runs fast in the child-spawnig type reduction. But the result denied the advantage of depth first strategy .

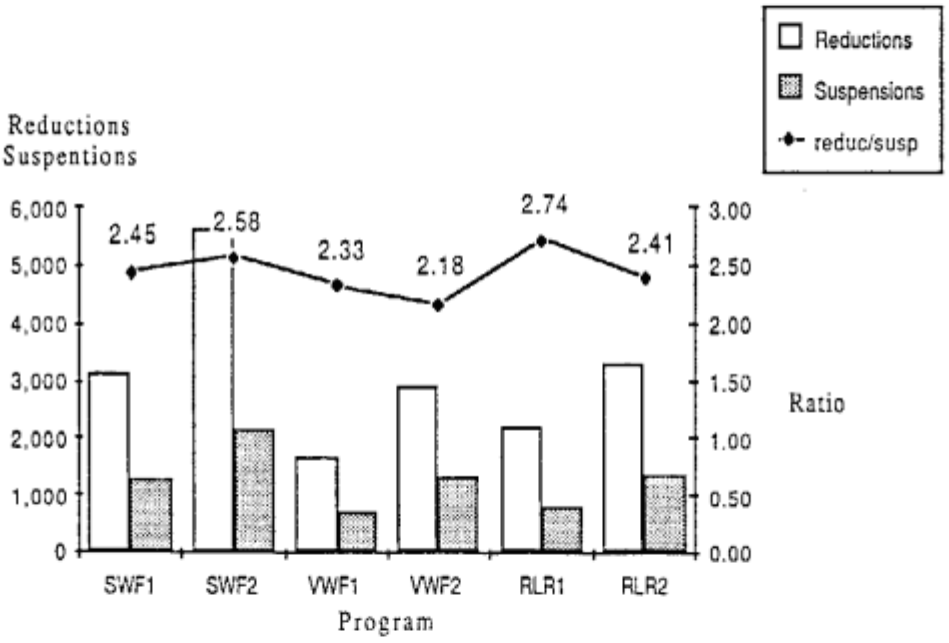
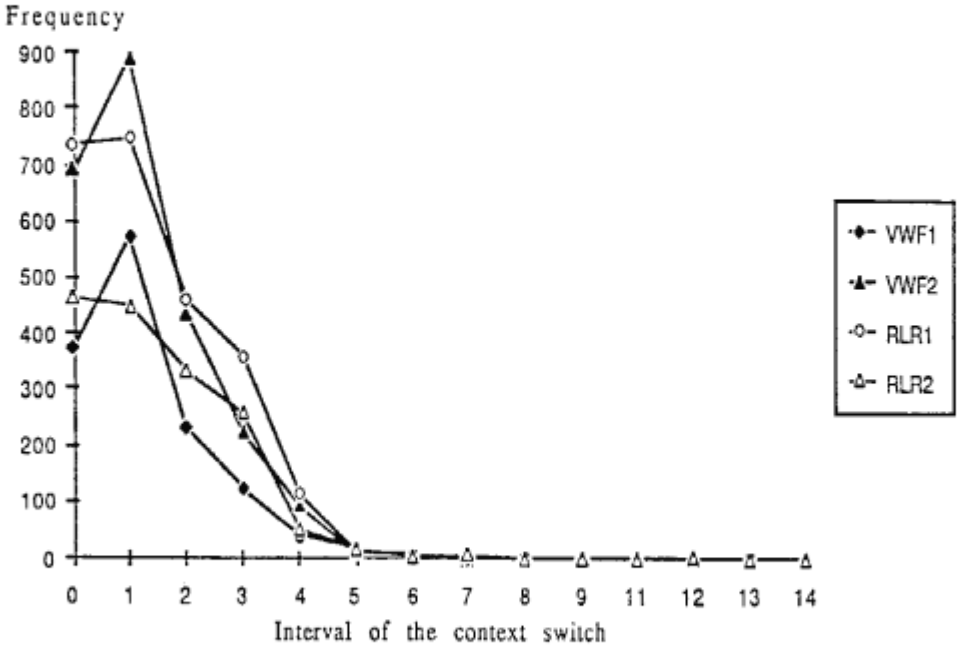
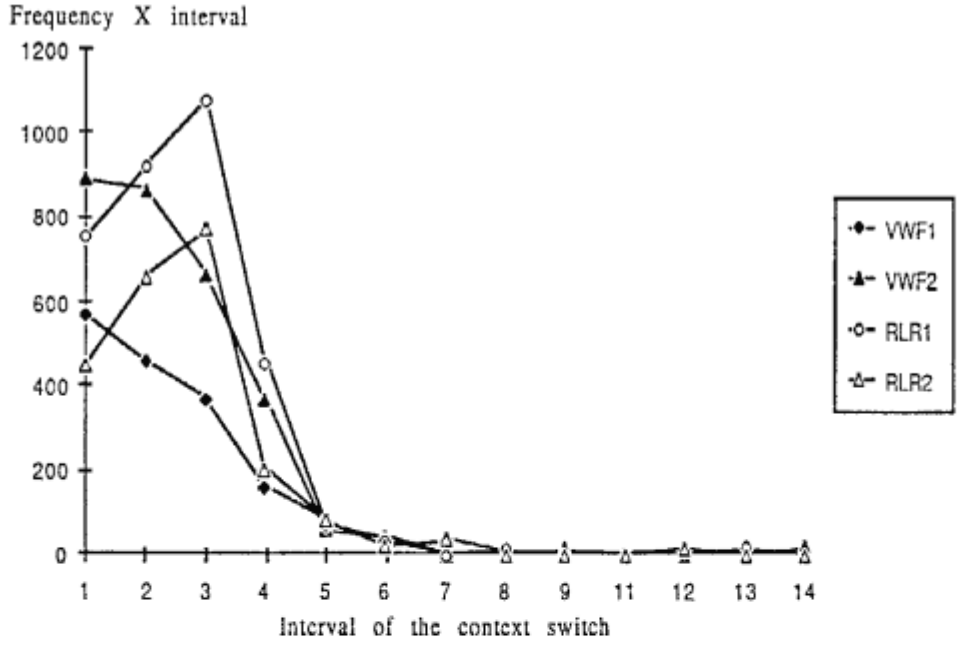


Figure 10. Number of reductions and suspensions

We measured intervals between context switches. A histogram of the intervals is shown in *Figure 11*. The raw data (*Figure 11* (a)) is the number of intervals, and the weighted data (*Figure 11* (b)) is the product of the number of intervals and the interval itself, which expressed the number of reductions they include. This figure shows that the zero and one intervals are predominant.



(a) Raw data histogram



(b) Weighted data histogram

Figure 11. Histogram of context switch interval

For an interval of zero, a goal, picked from the ready-goal queue, is not actually executable. This is because goals are put into the ready-goal queue without checking the suspension condition, when the clause was committed.

We suggested two methods for decreasing the number of interval-0-type context switches. In our implementation, we adopted the second method since it decreases the number of context switches by 10%.

(a) Check suspension conditions when body goals are spawned

When a clause is committed, the suspension conditions of its body goals are checked. If these conditions are satisfied, the goal is spawned in the usual manner. If not, it is hooked on an uninstantiated variable and suspended immediately.

The conditions check suspends the greater part of the goals. However most of hooked goals are going to be executable and resumed until they are dequeued from the ready-goal queue. The method increases useless suspensions and resumptions. Thus we have not adopted this method.

(b) Giving priority to resumed goals

We thought the resumed goals have lower probability of suspension than the goals in the ready-goal queue. So, we give priority to the resumed goals. The resumed goal uses the high priority ready-goal queue and is reduced before the other goals in the normal queue. This method decreases the number of context switches at most 10% (*Figure 12*).

We had expected the maximum of context switch interval to exceed 100, and thus set up a bound of 100 reductions. However, we found the maximum is actually less than 40 in the parallel wire router programs.

The large number of context switches and the short interval of them indicate that the task size is small and its parallelism is fine grained. This fact requires that the PIM have two hardware supported functions. First, it must

have fast context switch support, and second, it must have an efficient goal management mechanism.

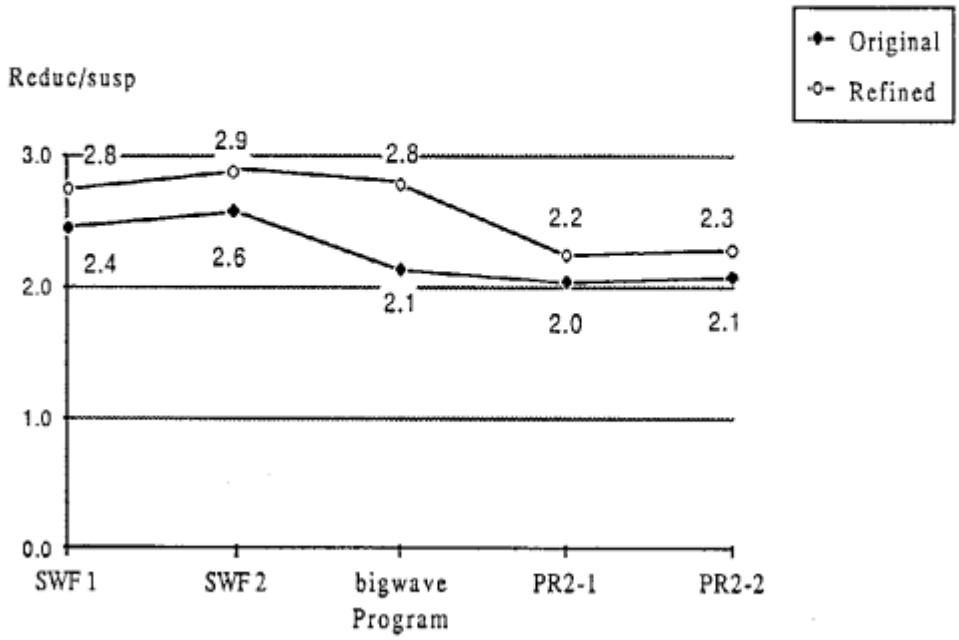


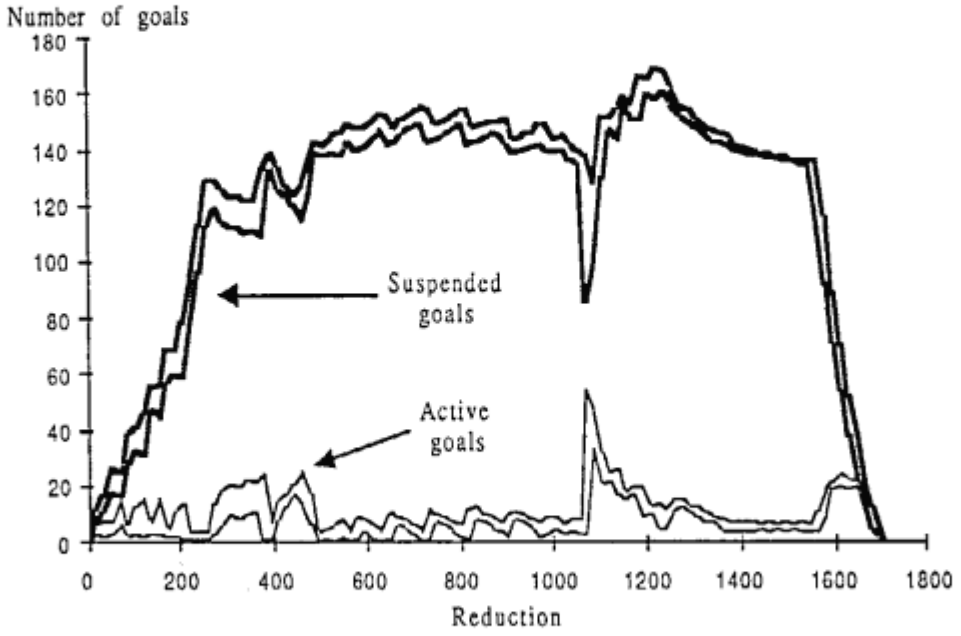
Figure 12. Refinement of suspension mechanism

4.3 Number of suspended goals

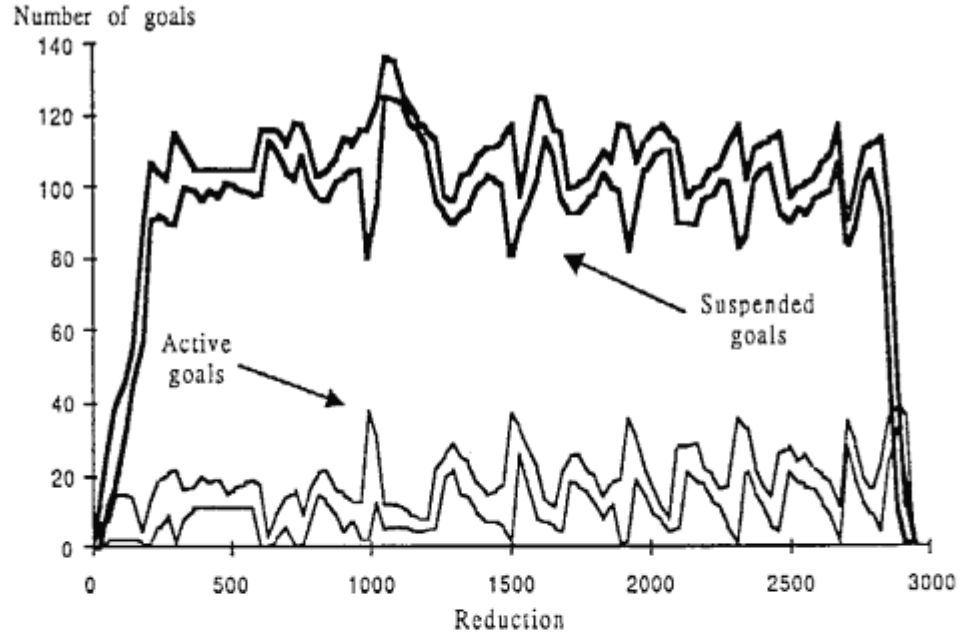
We measured the number of active and suspended goals. The active goals are ones in the ready-goal queue. The suspended goals are hooked at some variables. *Figure 13* shows the result. The horizontal axis is the time from the beginning measured with reductions. We summarize the dynamic characteristics of the parallel router program as follows.

(a) Suspended goals predominate

In all stages of execution, about 90% of goals are suspended. Active goals increase momentarily. The perpetual processes, the characteristic objects of parallel logic programming languages, are suspended almost their entire lives. Thus a non-busy-waiting strategy is essential for efficiently executing practical programs like our parallel wire router.



(a) Execution of the VWF1 program



(b) Execution of the RLR2 program

Figure 13. Number of active and suspended goals

(b) The number of goals does not fluctuate

Goals are divided into two classes, perpetual goals and temporary goals. The perpetual goals exist during all execution. When a perpetual goal

receives a message from neighboring goals, it spawns temporary goals to perform the task requested by means of the message. The temporary goals are almost active, and the perpetual goals are suspended. The number of temporary goals is negligible since there is a large number of perpetual goals.

In the initial stage, the number of suspended goals grows rapidly. In the intermediate stage, the summantion of active and suspended goals does not fluctuate. In the final stage, it diminishs abruptly and becomes zero. The pattern is shown in *Figure 13*.

At times during the intermediate stage, the resumption of a lot of perpetual goals appears as a big fall in the suspended goals and a sharp rise of the active goals. Even then, the total number of both goals does not fluctuate.

We had adopted depth-first strategy for our language processor to keep the number of goals small. According to *Figure 13*, however, the depth-first strategy also has a large number of goals. The depth first strategy does not have enough advantages we expected before.

5. Conclusion

We have developed an efficient FGHC interpreter. It runs from 40 to 80% of the speed of an efficient Prolog interpreter and from 10 to 40 times faster than an FGHC compiler with a busy-waiting strategy. It has enough performance for practical use.

At the same time, we choose the parallel wire router for practical parallel application programs and developed them in FGHC. According to the measurement and analysis of the parallel wire router, we pointed out the non-busy-waiting strategy is essential, but the depth-first strategy does not have enough advantages.

We are going to developing a multi processor implementation of language processor. And we also re-examine the language functions and the programming style of FGHC, because the tasks are too small to execute efficiently.

Acknowledgements

The authors wish to acknowledge the guidance received from the staff at ICOT, especially that of Mr. Kimura and Dr. Uchida, the laboratory's manager. Mr. Tanahashi and Mr. Hayashi of the Artificial Intelligence laboratory gave us the chance for our research and encouragement. We also wish to thank the other members of the laboratory staff for useful discussions, and Mr. Kishishita, Mr. Takaya and Mr. Hirano of FUJITSU SSL for cooperation in developing the parallel wire router programs.

References

[1] Goto A. and Uchida S. *Toward a High Performance Parallel Inference Machine - The Intermediate Stage Plan of PIM -*. TR 201, ICOT, 1986

[2] Ueda K. *Guarded Horn Clauses*. TR 103, ICOT, 1985.

[3] Clark, K.L. and Gregory, S. *PARLOG: parallel programming in logic*. ACM Transactions on Programming Languages and Systems 8, January 1986

[4] Shapiro, E.Y. *A subset of Concurrent Prolog and its interpreter*. TR 003, ICOT, Tokyo, February, 1983

[5] Bowen D.L. (ed.), et al. *DEC system-10 Prolog User's Manual*. Dept. of Artificial Intelligence, Univ of Edinburgh, 1982.

[6] David H. D. Warren. *An Abstract Prolog Instruction Set*. Technical Note 309,Artificial Intelligence Center, SRI, 1983.

[7] Kimura Y. *An Abstract KL1 Machine and its Instruction Set*. to be submitted at '87 SLP.

[8] Burstein, M. and Pelavin, R. *Hierarchical Wire Routing*. IEEE Trans. Computer-Aided Design Integrated Circuits and Systems Vol. CAD-2, No. 4, Oct. 1983, pp. 223-234.

[9] Sindo, T. Kawato, N. Ishii, M. *'Parallel router*. Proc. National Conference of IECE, 1985, in Japanese.

[10] Ueda, K. Chikayama, T. *Concurrent Compiler on Top of Prolog* Symposium on Logic Programming, 1985 pp. 119-127