TR-230

A Distributed Implementation of Flat GHC on the Multi-PSI

by

N. Ichiyoshi, T. Miyazaki and K. Taki

March, 1987

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

A Distributed Implementation of Flat GHC on the Multi-PSI

N. Ichiyoshi, T. Miyazaki and K. Taki

ICOT
21F. Mita Kokusai bldg., 4-28, Mita 1, Minato-ku,
Tokyo 108, Japan

Abstract

We have implemented the parallel logic programming language Flat Guarded Horn Clauses (FGHC) on the Multi-PSI machine, which is a collection of Personal Sequential Inference Machines (PSIs) interconnected by a fast communication network. FGHC goals are distributed among the PSI machines to be executed in parallel. The key to our implementation are the introduction of the "proxy and foster-parent" scheme and the development of an inter-processor communication protocol to avoid racing between simultaneous operations on the distributed AND-tree. The objective of our current project is to test a parallel implementation of FGHC and to develop basic and application software.

1 Introduction

The Multi-PSI [6] is a collection of Personal Sequential Inference Machines [10] (PSIs) interconnected by a fast communication network with a two-dimensional mesh topology. It is intended to serve as a workbench for research in parallel logic programming before a full-fledged parallel inference machine (PIM) comes into being. Major projects concerning the Multi-PSI are:

- Design of a parallel logic programming language that will serve as the kernel of the Fifth Generation Computer System and its implementation on a parallel machine,
- Development of a prototype operating system for a parallel machine,
- Development of parallel application programs.

The evaluation of the Multi-PSI and parallel software on it is expected to influence the design of the PIM.

The contents of the paper are as follows.

Section 2 is an introduction to Flat GHC. It gives a rather informal semantics of the execution of Flat GHC goals.

Section 3 is on the abstract machine architecture that we assume in our distributed implementation of Flat GHC. The current Multi-PSI is a small-scale model of the architecture.

Section 4 contains the execution mechanism of Flat GHC goal on a single processor. We extend it to the multi-processor system in Section 5. The proxy-and-foster-parent scheme is central to our AND-tree handling operations in a distributed environment.

Our distributed unification algorithms are explained in Section 6 with several examples.

Finally the current status of our implementation and the future directions are discussed in Section 7.

Brief descriptions of the types of cells (terms) will be found in Appendix A. In the subsequent sections, when we say a cell X is of an **undef** cell, this means the cell X is of type **undef**, and so on.

We list the inter-PE messages and brief explanations in Appendix B.

2 Flat GHC

Guarded Horn Clauses (GHC) [7,8] is a parallel logic programming language similar to Concurrent Prolog [5] and PARLOG [3].

A GHC procedure consists of a set of clauses of the form:

$$\underbrace{H :- G_1, \ldots, G_m}_{\text{guard}} \mid \underbrace{B_1, \ldots, B_n.}_{\text{body}} \quad (m > 0,\ n > 0)$$

where H, G_i, and B_i are atomic formulas. H is called the head, G_i the guard goals, B_i the body goals. The vertical bar (|) is called a commitment operator.

The execution of a GHC procedure can be intuitively explained as follows. When a procedure is called, all clauses defining the procedure can run in parallel. If some of the clauses succeed in the execution of the guard part, one and only one of them is (nondeterministically) selected and execution of its body part begins (the execution of the

other clauses is discarded). This is called a reduction of a goal into body goals. The unification in the guard part cannot instantiate variables in the caller's environment – instead the unification is suspended until the variables become instantiated. This is the basic mechanism of synchronization in GHC.

Flat GHC (FGHC) is a subset of GHC where the guard part of the clauses contains unifications and calls to system predicates but no calls to user-defined predicates. (The repertoire of the system predicates is implementation dependent.) This restriction makes FGHC free of nested guards and allows an efficient implementation.

We extend the original FGHC to include the metacall mechanism [3] and the pragmas [4,1]. The former is included for ease of writing system programs.

The pragmas are designed to allow the programmer explicitly to specify how the goals should be assigned to the processors. We allow body goals to have pragmas specifying on which processing element the goal should be executed when the parent goal is reduced to the body. Syntactically a goal `G` with a pragma `P` is denoted by

```
P@@G.
```

Currently in our system a pragma is just an integer that directly specifies the processor number. In the following example, the invocation of *translate(1)* will result in the first body goal to be executed on the processor **PE#1**, the second on **PE#2**, and the third on **PE#3**.

```
translate(PE1) :-
    PE2 := PE1+1,
    PE3 := PE1+2 |
        PE1@@instream(I),
        PE2@@translate(I,O),
        PE3@@outstream(O).
```

3 The Abstract Machine Architecture

The architecture of the underlying machine that we assume is a collection of processing elements (PEs) interconnected by a fast network. More specifically, we assume the following:

1. PEs are assigned unique identification numbers (**PE#1**, ...) to be used in inter-PE communication and inter-PE data references.

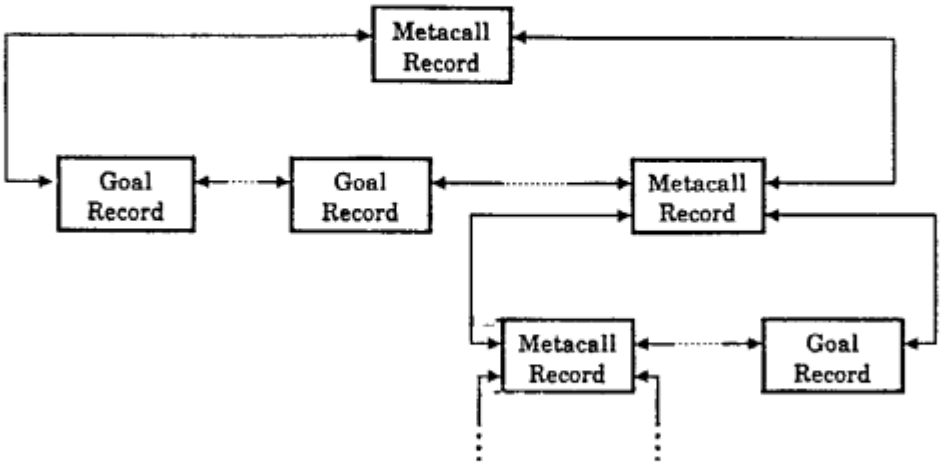

Figure 1: AND-tree

2. Each PE has a tagged architecture and is capable of executing FGHC goals by itself.
3. PEs have separate local memory spaces, but each PE can access external data by the inter-PE reference mechanism.
4. PEs communicate with each other by message passing via the network.

The reason for choosing the non-shared memory architecture is to allow a large-scale system where processors cannot be tightly coupled. Also the separation of intra- and inter-PE addressing systems has two advantages over the simple linear global address space. One is that local and global garbage collections can be separated. The other is that the size of the global address space will not be restricted by the internal addressing mechanism of the constituent PEs. Although the Multi-PSI — collection of PSIs, which are full workstations themselves — is not expected to become such a large-scale system, we have employed the above architecture to study load sharing mechanisms, etc., that will be used in a future highly parallel inference machine.

4 Intra-PE Processing

4.1 The AND-tree

The AND-tree (Figure 1) maintains all goals under execution. The roots of the subtrees are metacalls, and the leaf nodes are FGHC

goals other than the metacalls. (Probably the root metacall will be the one invoking the operating system.)

A metacall is of the form

```
call(Goal,Result,Control),
```

where `Goal` is the goal to be executed under it, `Result` is the result of the call (one of *success*, *failure*, *stop*), and `Control` is an input stream through which control messages (sequence of *suspend*, *resume*, *stop*) pass.

The leaf goals are direct or indirect descendant goals of the metacall just above them in the AND-tree. Note that if `Goal` fails, the metacall does not fail but instantiate `Result` to *failure* [1].

We employed the metacall mechanism for the following reasons:

1. We intend to write an operating system in FGHC where user programs can fail but the system as a whole must not fail. We are not taking the approach of the metainterpreter or source-to-source transformation right now.
2. Also in system programs, resource management will require the concept of a "job" or a "task" which the metacall mechanism can naturally introduce.

The metacalls and the goals are represented by the metacall records and the goal records, respectively. A metacall record has fields for the metacall identification number, the links to the descendant goal records, the status of execution, a pointer to the code, allocation information, etc. A goal record's fields are those for the links to sibling goal records, the arguments, the status of execution, a pointer to the code, etc.

4.2 Execution of goals

The PE has prioritized queues of executable goals (*ready goal queues*). At the beginning of a processing cycle (called a *reduction cycle*) the scheduler serves the first goal in the queue with the highest priority. At the beginning of each reduction cycle the scheduler also attends to the messages that have arrived during the previous cycle.

When a goal is scheduled, its arguments in the goal record are moved to the argument registers, and the control transfers to the code for the procedure. The code is a modified Warren Abstract Machine code[9]

with extensions of guard part unification and body reduction instructions. The clauses for the procedure are tried sequentially in the textual order (top to bottom) until

1. the guard of some clause has succeeded, in which case the clause is selected and the body is expanded, or
2. all clauses have failed or have been suspended.

The body is expanded in the following manner. Unifications if any in the body are not put into the ready goal queues but executed *in line*. How body part unification is done is described in Section 6. If the unification succeeds, the body expansion continues as follows. If the body contains no goals other than unifications, the parent goal is just removed from the AND-tree (successful termination of the parent goal). Otherwise all the goals except one are enqueued at the ready goal queue with the appropriate priority, and after setting the argument registers, control transfers to the goal that was not enqueued. Thus the next reduction cycle omits the dequeueing of a goal. This optimization which corresponds to tail recursion optimization (TRO) in Prolog reduces the number of expensive context switches.

When, during the trial of guards, some passive part unification leads to suspension, a pointer to the variable responsible for the suspension is pushed onto a special stack called the *trail stack*, and control moves on to the next clause. (The trail stack is set to empty at the beginning of every reduction cycle.)

If the clauses have been exhausted without success of any of the guards, there are two possibilities:

Case-1: the trail stack is empty, or

Case-2: there are pointers to variables responsible for suspension of clauses.

Case-1 means that all guards have failed, which means the goal has failed. This causes the parent metacall to terminate with the result of failure.

In **Case-2** the goal is entered into the suspension queues (described later) of the variables. When one of those variables becomes instantiated, the goal is enqueued at the tail of a ready goal queue. The trail stack is ignored when one of the guards has succeeded, thus eliminating unnecessary handling of suspension queues. This is exactly why the trail stack is introduced.

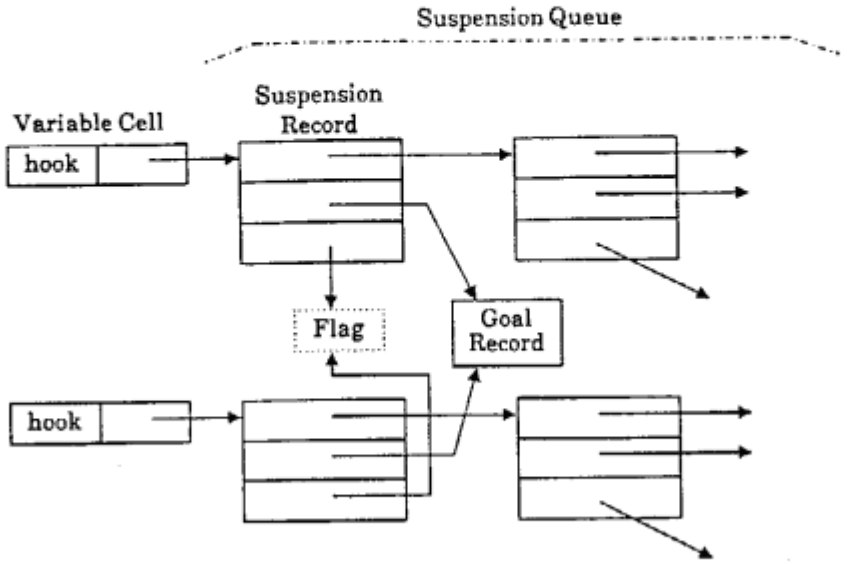

Figure 2: Suspension Queue

4.3 Suspension and resumption

Every uninstantiated variable causing suspension of clauses keeps a list of goals that invoked those clauses. This list is called the *suspension queue* of that variable. A goal in a suspension queue of a variable is said to be *hooked* to the variable, and the variable is called a *hook variable*. The hook variable is a data type distinct from that of uninstantiated variable in the implementation, though they are logically equivalent. (An uninstantiated variable can be thought of as a hook variable with an empty suspension queue.) Actually a suspension queue is a chain of suspension records (Figure 2), whose fields are:

next record: points to the next suspension record (or nil).
goal record pointer: points to the suspended record.
flag pointer: points to a flag record shared by those variables whose instantiation the suspended record is awaiting.

Generally more than one clause called by a goal may be suspended on different variables. In this case, one suspension record is created for each variable. These suspension records share one flag record which is initially *off*. The goal is resumed when one of those variables get instantiated. When this happens, the flag record is set to *on* to prevent the variables instantiated later from waking up the goal more than once.

5 Inter-PE Processing

5.1 Memory management

Each PE has its own local memory space. In one PE, there are two kinds of references: internal and external. An internal reference is a simple reference to (represented by logical address of) a cell in the local memory. An external reference consists of a triple of a read flag, the processor number and the cell identification number. The processor number together with the cell identification number uniquely determines the referenced cell. The read flag is used for queueing read requests for the referenced cell. A cell in a PE which is referenced from outside the PE is said to be *exported*.

Each PE maintains an *export table* to retrieve exported cells when their cell identifition numbers are given as the key. We employ this indirect addressing mechanism in favor of the representation of external reference by a pair of the PE number and the internal address to make local garbage collection in a PE possible without pointer updates outside of it — only the export table must be updated when the cells have been moved.

5.2 Inter-processor handling of goals

In a multiprocessor system the logical AND-tree crosses PE boundaries. If we simply represented it using external reference links, the rate of inter-processor tree operations could be very high and the synchronization would be very complicated. We devised a *proxy-and-foster-parent scheme* to avoid this (Figure 3). The idea is to separate tree operation from the the execution of goals.

In the proxy-and-foster-parent scheme, descendant goals of a metacall can be executed in any PE. In one PE, all goals with the same parent metacall are maintained under a metacall record or a *foster-parent record*. A metacall record resides in the PE, say **PE#i** where the metacall was issued, while foster-parents reside in PEs other than **PE#i**. They are given the same metacall identification number as the metacall record. (Metacalls are distinguished by their metacall identification numbers.) A foster-parent communicates with a *proxy record* under a metacall record or a foster-parent record in another PE, as described later.

The physical tree structure with the metacall record as its root is

5.2 *Inter-processor handling of goals*

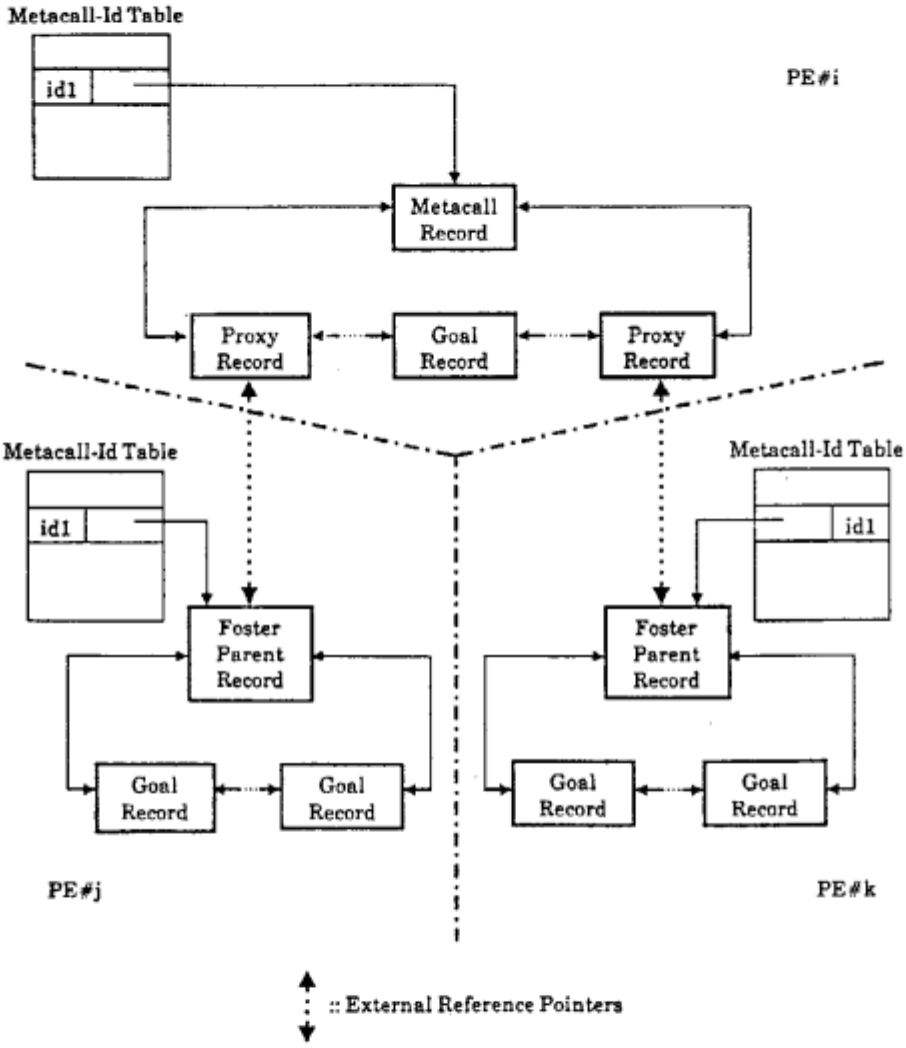

Figure 3: Distributed AND-tree

distributed among the PEs. The life time of a proxy record covers that of the corresponding foster-parent record to guarantee that this dynamic tree structure be always well-formed. A foster-parent serves as the cache of the metacall record, while a proxy record represents those goals under the foster-parent with which it communicates. AND-tree operations are propagated upward and downward through the proxy to foster-parent links.

Each PE has a *Metacall-Id* table with which to retrieve metacall records and foster-parent records from the metacall identification numbers.

We will describe the operations on the AND-tree at various events.

- Throwing of a goal

  When the body of a clause is expanded, the PE executing the goal can request other PEs of the execution of some of the body goals. This is called *throwing of goals*. The throwing of goals is specified by the programmer by attaching pragmas. Goal throwing is done in the following manner.

  A PE (**PE#i**) can request another PE (**PE#j**) of the execution of a goal by sending a *throw_goal* message to that PE. When **PE#i** does this, it first creates a proxy record for the goal and send a throw_goal message carrying the goal, the external reference to the proxy record, and the metacall identification number. It does not matter whether the parent goal was running under a metacall record or under a foster-parent record.

  On receiving the message, **PE#j** searches the Metacall-Id table to see if there is already a metacall record or foster-parent record having the same metacall identification number as the one in the message. If there is one, **PE#j** puts the goal under it and sends back a *cancel* message to **PE#i**. If there is none, it creates a foster-parent record with the given metacall identification number and enters it in the Metacall-Id table. **PE#j** then puts the thrown goal under the foster-parent record, and sends back to the **PE#i** a *ready* message bearing the external reference to the foster-parent record.

  In either case **PE#j** accepts the execution of the goal. The difference is that **PE#i** will erase the proxy record when it receives a cancel message whereas it will not when it receives a ready messages. In this way, no PE will have more than one metacall or

foster-parent record for a given metacall number, and the foster-parent to proxy record links will not contain a cyclic chain.

- Successful termination of a goal

  When a goal succeeeds, the goal record for it is removed from the physical subtree whose root is either a metacall record or a foster-parent record. If only the root is left after the removal,

  - if the root is a metacall record, it instantiates the result to *success* and terminates.
  - if the root is a foster-parent record, it sends a *terminate* message to the corresponding proxy record, which in turn succeeds just like a normal goal (propagation of success upward).

- Failure of a goal

  If a goal fails, it causes the subtree containing it to fail. All goals under the root record are killed. If there are proxy records, *kill* messages are sent to the corresponding foster-parent records, propagating failure downward. Metacalls under the root are also killed. If the root is a metacall record, the result is instantiated to *failure*. If the root is a foster-parent record, a *fail* message is sent to the corresponding proxy record causing it to fail with the same effect as the failure of a goal record (propagation of failure upward).

- Spawning of child goals

  When a goal is reduced to its child goals upon reduction, the parent goal record is replaced by the child goal records in the subtree. Some of the goals can be thrown outside of the PE.

6 Unification

In FGHC there are two kinds of unification. One is unification in the guard part (passive part unification) and the other is unification in the body part (active part unification).

We will sketch our algorithms for passive and active part unifications, focusing on how inter-PE unification is realized.

6.1 Passive part unification

Table 1 summarizes the actions taken in the passive part unification of X and Y for all combinations of cell types of X and Y.

Suspend is suspending the unification and pushing a pointer to the uninstantiated variable responsible for suspension onto the trail stack.

Table 1: Passive Part Unification

X \ Y	undef	exref	constant	structure
undef	suspend	suspend	suspend	suspend
exref	suspend	read_value†	read_value	read_value
constant	suspend	read_value	X==Y	fail
structure	suspend	read_value	fail	unify_struct(X,Y)‡

† A read_value message is sent to obtain the value of one of the external references. When the goal is resumed by the return of the value, another read_value message is sent for the value of the other external reference.

‡ Unification of two structures X and Y consists of checking the equality of the principal functors and recursively unifying all elements of X and Y.

Read_value is the case where the value of externally referenced cell needs to be obtained. This is treated like suspension, because in either case unification cannot continue until the needed value is obtained. A pointer to the **exref** cell is pushed onto the trail stack and the PE, say **PE#i**, sends a *read_value* message to the PE, say **PE#j**, housing the externally referenced cell. On receiving the message, **PE#j** checks if the referenced cell is already instantiated. If it is, **PE#j** will return the value by sending a *return_value* message back to **PE#i**. If the value of the referenced cell is an external reference, **PE#j** simply passes a read_value message to the referenced PE. Otherwise, it is an **undef** or **hook** cell, and a special goal for returning the value is hooked onto the referenced cell. This goal will be woken up when the cell is instantiated and return the value to **PE#i**.

In returning a nested structure, **PE#j** can be either eager (returning the whole structure), lazy (returning only the surface level with embedded external references to the substructures) or in between (returning a partial structure with a certain level of nesting). Which of the above is more efficient depends on how the returned value is to be used. Though a lazy returning (or returning up to the second level) seems to be a safe bet, we may have to give the programer control over eagerness/laziness.

6.2 Active part unification

In a distributed environment a number of processors can work on the same structure at the same time. Racing will happen if more than one processor tries to write into one uninstantiated variable simultaneously. Locking of variables could solve this problem, but would introduce a new problem of deadlock avoidance. We have solved this problem by prohibiting the PE from explicitly writing (instantiating) variables in another PE. Suppose for example **PE#i** has an external reference to a variable X in **PE#j** and wants to unify it with an atom *ok* in the body. Instead of first checking if X is uninstantiated and then asking **PE#j** to instantiate X to *ok*, **PE#i** simply sends a *unify* message to **PE#j** asking to unify X with *ok* and **PE#j** will answer whether the unification has succeeded or failed. The unify message can be considered as a special case optimization of the throw goal message. We have also introduced an ordering of PEs to avoid loops in external reference chains. Specifically, in unifying two uninstantiated variable cells in two distinct PEs, the external reference must always be made from the cell in a PE with a smaller PE number to the cell in a PE with a larger PE number.

The following simple examples illustrate how unification involving external references is done. Let **PE#i** be the processor where the unifications are initiated. Superscripts over the variables represent the cell types.

Example 1. $X^{exref} = f(A^{undef})$

Suppose the PE number field of X is **PE#j**, and the cell identification number field is *cidx*. First, a proxy record is created for the unification. Second, a unification message is sent to **PE#j** with *cidx*, *f(A)*, and other necessary data.

The representation of *f(A)* needs some explanation. Since exporting *f(A)* involves exporting the **undef** cell A, A is entered into the export table as the entry number *cida* (which becomes A's cell identification number). The structure *f(A)* is represeted in the message as *f*(**exref(PE#i,***cida***)**). In the case of a nested structure, the data is copied in the return_value message up to a predetermined copy level and the deeper substructures are represented by external references to them.

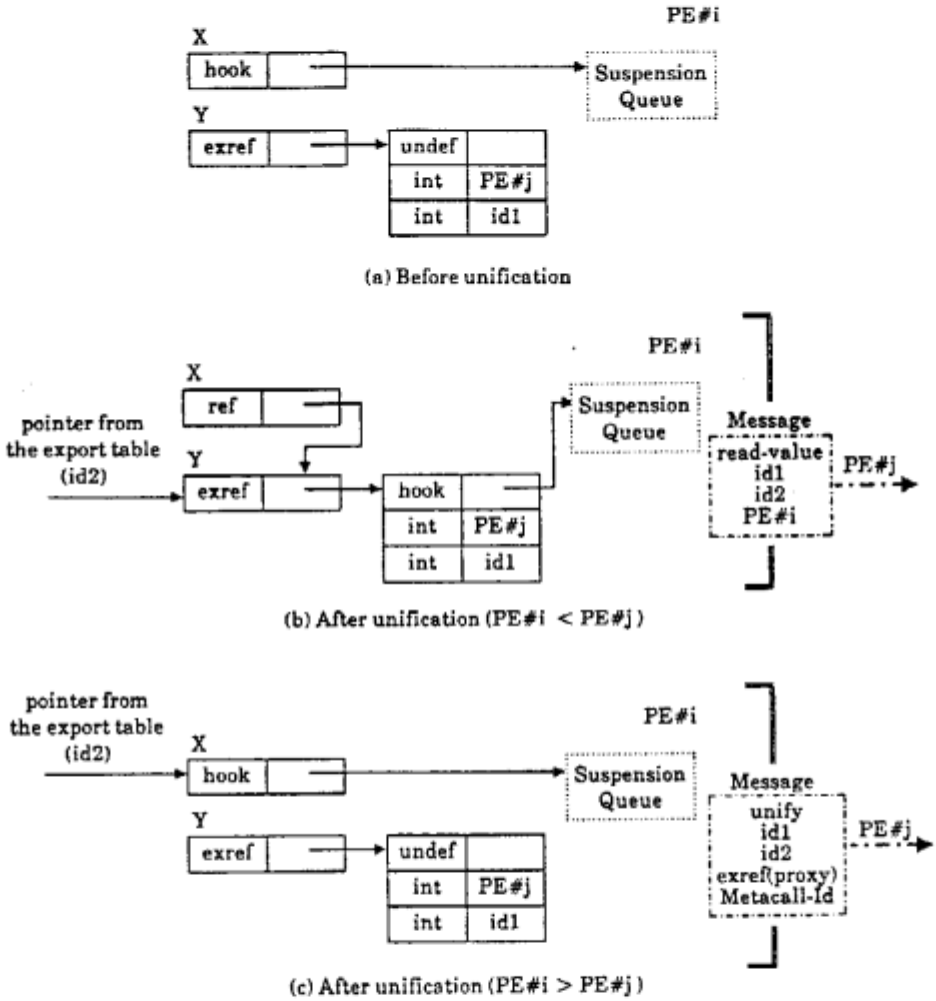

Figure 4: Active Part Unification

6.2 *Active part unification*

On receiving the unify message, the **PE#j** tries to execute the goal *X=f(A)* without bothering to create a goal record and to enqueue it to a ready goal queue, since unification is often very simple. If X is an **undef** or **hook**, it is instantiated to *f(A)* and a cancel message is sent back to the original PE to erase the proxy record. If it is an **exref** to a cell in **PE#k**, the unify message is simply passed on to **PE#k**. Otherwise, the requested unification is treated as a thrown goal.

Example 2. $X^{hook} = Y^{exref}$

Figure 4a shows the states of X and Y before unification. Suppose Y refers to a cell Z in **PE#j**. If i is smaller than j, X can become an external reference to Z by the ordering rule. But because X is a hook cell, a measure has to be taken so that the instantiation of Z may be propagated to X for the goals awaiting the instantiation of X to resume. Thus a read request of the value of Y is made in the same way as in passive part unification, the suspension queue of X is appended to that of Y's read flag, and the cell X becomes (is overwritten by) an internal reference to Y. When the value of Z is returned, the goals awaiting for X to be instantiated as well as the goals having requested the value of Y are resumed. Figure 4b shows the situation just after the read_value message has been sent.

If i is larger than j, X must not become a reference to Y by the PE ordering rule. A proxy record for the unification is created and a message is sent to **PE#j** requesting unification of X (an external reference to X) and Z. **PE#j** will treat the unify message in a similar way as Example 1. The situation just after the unify message has been sent is shown in Figure 4c.

One major difference between passive and active part unifications in our scheme is:

- In passive part unification, unfication procedure itself is exclusively done on the PE initiating the unification. Other PEs just return values of external references upon request.
- In active part unification, unification procedure can become really distributed: unification between two structured terms can trigger unify messages to be sent to a number of PEs for substructure unification.

7 Current Status and Future Directions

Our FGHC system runs on a 6-processor configuration Multi-PSI. Currently all constituent machines must have a FGHC compiler on them. To run a FGHC program, one has to distribute the source code to all PEs to be compiled on each of them. FGHC clauses are compiled into the abstract machine instructions each of which is actually a method call in an emulator program written in ESP [2]. Before starting the program, the atom table (table of atom names appearing in the program and thier identification numbers) is distributed to all PEs. This is because atoms are represented by common identification numbers in inter-PE messages. After the above has been done, execution can be initiated at any PE.

We have tested a number of sample programs including qsort, the N-queens problem and the knapsack problem, and started to gather statistics on the numbers of reductions, messages sent, suspensions, etc. Global garbage collection is not yet implemented. (Local garbage collection is simply garbage collection on each constitutent PSI.) We plan to implement it when the Multi-PSI verson II is delivered. It will be based on PSI-II, a smaller, faster version of the current PSI. FGHC abstract machine code will be directly supported by the firmware of the Multi-PSI version II.

Software issues we will concern ourselves will be:

1. Development of a rudimentary user environment.

   We have to write system service programs including input/output, code management (such as dynamic code loading).

2. Load balancing

   We are considering two ways of doing this:

   - pragmas

     This is to let the user specify how the goal should be assigned to the PEs. Currently, the pragmas specify physical PE identification numbers on which the goal should be executed. In the future we will let the user specify load distribution at some logical level without considering physical configurations. [6] proposes one such mechanism.

   - automatic load balancing

     This is to have the system dynamically detect load imbalance and correct it, although simple averaging of computational load

can lead to lower performance due to increased inter-processor communication. Perhaps localization of inter-PE communication should be controlled by user-specified pragmas and the logical-to-physical mapping should be managed by the system through dynamic load balancing.

3. Deadlock detection

   In a programming language like FGHC where one can easily write programs that will be suspended forever (e.g. a spelling mistake of a variable in a producer process can make a consumer process wait for the instantiation of a variable that will never be instantiated), deadlock detection is almost vital to program debugging. Efficient deadlock detection mechanism must be implemented.

Acknowledgements

We thank the members of the FGHC implementation group at ICOT 1st and 4th Laboratories and collaborating companies, and Dr. K. Furukawa and Dr. S. Uchida, heads of ICOT 1st and 4th Laboratories for valuable discussions and encouragement.

Notes

[1] A metacall can fail if the instantiation of `Result` fails because of earlier instantiation by some other goal of that same variable to something else. But in practice this should not happen.

References

[1] T. Chikayama. Load balancing in a very large scale multi-processor system. In *Proceedings of Fourth Japanese-Swedish Workshop on Fifth Generation Computer Systems*, SICS, 1986.

[2] T. Chikayama. Unique features of ESP. In *Proceedings of FGCS'84*, ICOT, 1984.

[3] K. L. Clark and S. Gregory. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[4] E. Shapiro. Systolic programming: a paradigm of parallel programming. In *Proceedings of The International Conference on New Generation Computer Systems 1984*, pages 458–470, 1984.

[5] E. Y. Shapiro. *A Subset of Concurrent Prolog and Its Interpreter.* ICOT Technical Report TR-003, ICOT, Tokyo, Japan, January 1983.

[6] K. Taki. The parallel software research and development tool: Multi-PSI system. In *Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium 86*, pages 365-381, 1986.

[7] K. Ueda. *Guarded Horn Clauses.* Technical Report TR-103, ICOT, 1985.

[8] K. Ueda. *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard.* Technical Report TR-208, ICOT, 1986.

[9] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Techinical Note 309, SRI International, Oct. 1983.

[10] M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida. *The Design and Implementation of a Personal Sequential Inference Machine: PSI.* ICOT Technical Report TR-045, ICOT, 1984. Also in New Generation Computing, Vol.1 No.2, 1984.

Appendices

A Data types

The data types of the cells include the following:

constant: an atomic constant. There can be interger, atom, string, and other data types.

structure: a structured term. Lists and streams can be distinct data types.

undef: an uninstantiated (unbound) variable.

hook: a hook variable. It has a pointer to the suspension queue.

ref: an internal reference (internal pointer).

exref: an external reference. Points to a record consisting of the read flag, the PE number and the cell identification number. The last two uniquely locate the referenced cell in the machine. In the text, we refer to the read flag of the record pointed to by the **exref** cell X, etc. as the read flag field of X, etc.

B Inter-PE messages

In the following we list the kinds of messages passed between the PEs. A message packet contains the message plus the identification numbers of the sender and receiver PEs, **PE#x** and **PE#y** respectively.

`read_value(Cid, Xref)`: requests **PE#y** to return the value of the exported cell with cell identification number *Cid* when or if it is instantiated. The value is to be returned to *Xref* in **PE#x**. There will be at most one return_value message in reply for this message.

`return_value(Xref)`: returns the value of an exported cell. *Xref* is the external reference to the **exref** cell whose value had been requested.

`throw_goal(Goal, Metacall-Id, Proxy)`: requests **PE#y** to execute the goal *Goal* under a metacall identification number of *Metacall-Id*. *Proxy* is the identification number of the proxy record in the export table of the sender PE.

`unify(Cid, Y, Metacall-Id, Proxy)`: requests **PE#y** to unify the exported cell with cell identification number *Cid* with *Y* as a goal with *Metacall-Id*. *Proxy* is the external reference to the proxy record created in **PE#x** to to represent this unification.

`ready(Proxy, FosterParent)`: notifies *Proxy* in **PE#y** in acknowledgement to a throw or unify message that a new foster-parent record has been created with the given metacall identification number and its external reference is *FosterParent*.

`cancel(Proxy)`: notifies *Proxy* in **PE#y** in acknowledgement to a throw or unify message that no new foster-parent record has been created because there had already been a metacall record or a foster-parent record with the given metacall identification number.

`terminate(Proxy)`: notifies *Proxy* in **PE#y** that all the goals it represents have successfully terminated.

`fail(Proxy)`: notifies *Proxy* in **PE#y** that one of the goals or unifications it represents has failed.

`kill(FosterParent)`: orders **PE#y** to kill the foster-parent whose external reference is *FosterParent* and all the goals being executed under it. The foster-parent is erased from the Metacall-Id table.

`dead(Proxy)`: notifies *Proxy* in **PE#y** in acknowledgement to a kill message that the kill operation has been completed. When this message is received, *Proxy* is erased.