TR-473

Research and Development of the Parallel Inference Machine in the FGCS Project

by
A. Goto

April, 1989

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

Research and Development of the Parallel Inference Machine in the FGCS Project

Atsuhiro GOTO*
Institute for New Generation Computer Technology (ICOT) †

Abstract

As part of the FGCS project, we are developing parallel inference machine (PIM) systems based on a logic programming framework. The research and development of PIM includes the PIM hardware architectures and the parallel implementation of KL1.

KL1 has been designed as the PIM kernel language, so that both PIM applications and the parallel operating system (PIMOS) can be written in KL1. The characteristics of KL1 are used to solve KL1 parallel implementation problems, such as distributed resource management, goal scheduling and distribution, memory management, and distributed unification. They have been condensed into the abstract machine instruction set, KL1-B.

In designing the hardware architecture of the PIM pilot machine, a hierarchical configuration has been introduced to connect more than 100 processing elements. A new instruction architecture for KL1 is provided for the processing elements. A coherent cache protocol has been designed to make high-performance clusters, each of which includes eight processing elements connected with shared memory. These clusters will be connected by a multiple hyper-cube network.

1 INTRODUCTION

The research and development (R&D) of the parallel inference machine (PIM) is one of the most important targets in the FGCS project. The PIM will be the pioneer of parallel processing in knowledge information processing system (KIPS) application fields.

The principal aim of parallel processing is to increase the execution performance so that users will be able to solve big application programs. The PIM should have many more features than conventional general-purpose machines. For example, pattern matching operations are important in many KIPS applications. However, it is insufficient to increase the efficiency of only limited functions in KIPS applications. In other words, development of the PIM should strive to develop more general and powerful machines than conventional ones. The PIM should also cover the functions of conventional computers, because AI machines are not simply game tree searching machines.

During the initial stage (1982 to 1984) of the FGCS project, the elementary mechanisms of the PIM were studied from various standpoints [12]. One of our most important policies in the R&D of the PIM system is to build up a total system based on logic programming, so that the system designers of the PIM can easily look through all levels of the system in a logic programming framework. This is an important way to solve the *semantic gap* argument: application and implementation are closer, therefore execution is faster. Therefore, the R&D of the current PIM, started in 1985 [13, 11], is being conducted with the design and implementation of the kernel language (KL1) and the PIM operating system (PIMOS) [6], which is written in KL1 as a *self-contained* operating system. We set the following

*CSNET: goto%icot.jp@relay.cs.net, UUCP: ihnp4!kddlab!icot!goto
†21F, Mitakokusai Building, 1-4-28, Mita, Minato-ku, Tokyo, 108, JAPAN

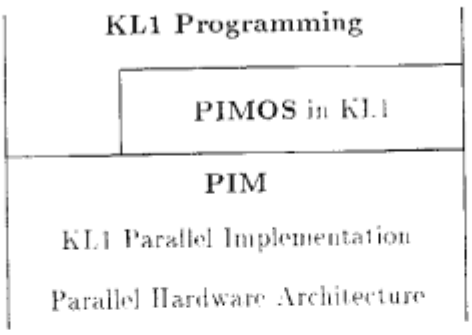

Figure 1: Parallel Inference Machine System Overview

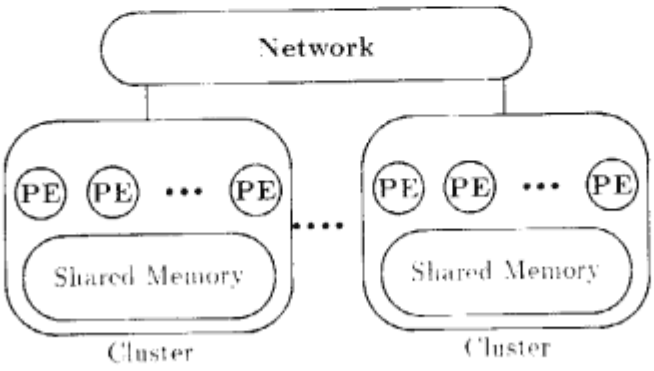

Figure 2: Abstract PIM Configuration

goals aiming at the PIM system shown in Figure 1.

First, we hope to realize a very high execution performance for logic programming in KL1. We believe that more than 100 times the performance of current machines will be necessary to enhance logic programming application research. Next, we aim to build practical systems that will be available as research tools in the final stage of the project. This is essential for application research. In addition, the development of total and practical systems stresses the importance of memory management and program control in parallel processing systems, and it also reveals the hidden problems in parallel processing. Finally we are trying to build the PIM system using KL1. The language features of KL1 are used to the full in the parallel architecture design.

This report describes the KL1 parallel implementation and the hardware architecture of the PIM pilot machine. Section 2 introduces KL1 with its meta-programming capability using shōen. Section 3 describes scheduling of KL1 goals within a cluster. Section 4 describes inter-cluster distribution of goals and their management using the shōen facility. Section 5 discusses how to reduce the communication cost in inter-cluster distributed unifications. Section 6 describes the incremental garbage collection mechanism embedded in the parallel KL1 implementation. The above schemes for the KL1 parallel implementation are condensed into an abstract instruction set called KL1-B [16]. KL1-B interfaces PIMs and KL1, just as WAM [29] interfaces Prolog and sequential machines. Section 7 overviews the KL1 B features.

The hardware architecture of the PIM pilot machine is described in section 8. We introduced a hierarchical configuration into the PIM hardware architecture (shown in Figure 2), which is assumed in the discussions about KL1 parallel implementation. Several PEs form a *cluster* with a shared memory. These clusters are interconnected by a communication network.

The Multi-PSI [28] system has been built to enhance the research for the KL1 parallel implementation and the PIMOS design. The Multi-PSI is a collection of the PSI machines [18] connected by

fast mesh network [27]. Most of the KL1 implementation issues in distributed environments have been studied through the design of the Multi-PSI system [17].

2 The Kernel Language KL1

2.1 FGHC: Base of KL1

KL1 was initially specified as Flat Guarded Horn Clauses (FGHC) [23], taking efficient implementation into consideration. The major reasons for choosing FGHC as the basis for KL1 are as follows. GHC has clear and simple semantics as a concurrent logic programming language, by which programmers can express important concepts in parallel programming, such as inter-process communication and synchronization. In addition, FGHC is an efficient language, in the sense that we can specify the machine level language.

FGHC is a language based on Horn clauses of the form: $H :- G_1, ...G_m|B_1, ..., B_n$. where H is the head of the clause, G_i are guards, "|" is the commit, and B_j are the body goals. In FGHC, as in Prolog, procedures are composed of sets of clauses with the same name and arity. Unlike Prolog, there are no nondeterminate procedures. Execution proceeds by attempting unification between a goal (the caller) and a clause head (the callee). If unification succeeds, execution of the guard goals is attempted. These goals can only be system-defined built-in procedures, e.g., arithmetic comparison. If the guard succeeds, the procedure call "commits" to that clause, i.e., any other possibly good candidate clauses are dismissed. If the head or guard fails, another candidate clause in the procedure is attempted (if all clauses fail, the program fails). There is a third possibility however: that the call *suspends*.

FGHC restricts unification in the head and guard (the "passive part" of the clause) to be input unification only, i.e., bindings are not exported. Output unification can be performed only in the body part (the "active part"). These restrictions allow AND-parallel execution of body goals and even OR-parallel execution of passive parts during a procedure call (the implementation discussed here executes passive parts sequentially and executes body goals in a depth-first). Synchronization between processes is inherently performed by the requirement that no output bindings can be made in the passive part. If a binding is attempted, the call *potentially* suspends. If none of the clauses succeeds, and one or more potentially suspend, then the procedure call suspends (possibly on multiple variables).

When any of the variables to which an export binding was attempted are in fact bound (by another process), the suspended call is resumed. These semantics permit stream AND-parallel execution of the program, i.e., incomplete lists of data can be streamed from one parallel process to another in a producer-consumer relationship. For example, when a stream runs dry, the consumer receives the unbound tail of a list and suspends. When the producer generates more data, the consumer is resumed and continues processing the transmitted data.

2.2 Meta-programming Capability Using Shōen

Starting from FGHC, KL1 has been extended so that it has become a practical language with the features required for the PIMOS design[1].

In GHC and FGHC, all goals compose a logical conjunction, so that the failure of a certain goal causes a global failure. However, the relation between the operating system and user programs must be that of a meta-level program and object level programs, where the meta-level program controls or monitors the object-level programs. Therefore, it is necessary to introduce a meta-programming capability into KL1.

[1] Chikayama et al. [6] describe the system programming features in KL1.

The meta-programming capability of KL1 is realized by the shöen facility. While tail-recursively executed goals look like small-grain threads of control (*processes*), a shöen defines a larger-grain computational unit, i.e., the concept of a *job* or a *task*. It deals with execution control of programs, resource management and exception handling.

A shöen may include child shoens, so that we can see KL1 goals form a tree-like structure (shöen tree) whose nodes are shöens and whose leaves are KL1 goals. In this case, when the execution in an outside (or parent) shoen stops, all execution in an inside (child) shöen stops automatically. When the outside execution is restarted, inside execution is also restarted.

Computing resources can be managed in each shöen to avoid, for example, infinite execution of user programs. The management of computing resources is roughly implemented as how many goal reductions can be performed within a shoen. The inside shöen can consume the computing resources within the amount of the resources that the outside shoen has.

A shoen is created by a call to the built-in predicate *execute/6*:

execute(Goal, Control, Report, Min, Max, Mask)

Goal specifies the initial goal, that is, the predicate name and its arguments, to execute in the shoen. All forked goals from the given *Goal* belong to the same shoen. *Min* and *Max* are the minimum and maximum possible priorities of goal scheduling allowed in the shöen. (See section 3.2.)

Control and *Report* are the control and the report streams. The control stream is used to start, stop or abort the shoen from outside. The monitoring process can be informed of events within a shöen such as the end of execution and exceptions through the report stream. Exceptions that have occurred in the shöen or are delegated from one of the child shöens are reported as a message to the report stream. *Mask* is a bit pattern for determining which exceptions should be handled in this shöen. The monitoring process can substitute a new goal for the goal that has given rise to the exception. An important thing to note is that there is no failure in a shöen. Any kind of failure is treated as an exception. The logical conjunction between KL1 goals is maintained within each shoen. In other words, goals in a shoen do not form a conjunction with goals outside the shoen. With the above meta-programming capability, we can describe not only the PIM applications but also the PIMOS, which controls parallel processes.

3 Goal Scheduling

3.1 Goal Reduction by Register Machines

While any unification of KL1 can be done in parallel under the semantics of GHC [23], we did not adopt this fine-grained parallelism, but, instead, the parallelism between goal reductions. This is because (1) unifications are granules that are too small to implement in parallel, and (2) we can extract enough parallelism between goal reductions.

A set of candidate clauses for the same predicate is compiled to KL1-B code as shown in section 7, executed by a single thread of control from guard to body. No parallelism is expected within each goal reduction. Each passive and active unification can be done by discrete KL1-B instructions as register memory or register-register operations, so that we can expect optimization by the compiler such as in register allocation.

3.2 Goal Scheduling on the Processor

A goal can be a *ready goal* (RG), a *suspended goal* (SG) or a *current goal* (CG), as shown in Figure 3. The *ready* goals are linked into a list forming a *ready-goal-stack*. In principle, a current goal is popped

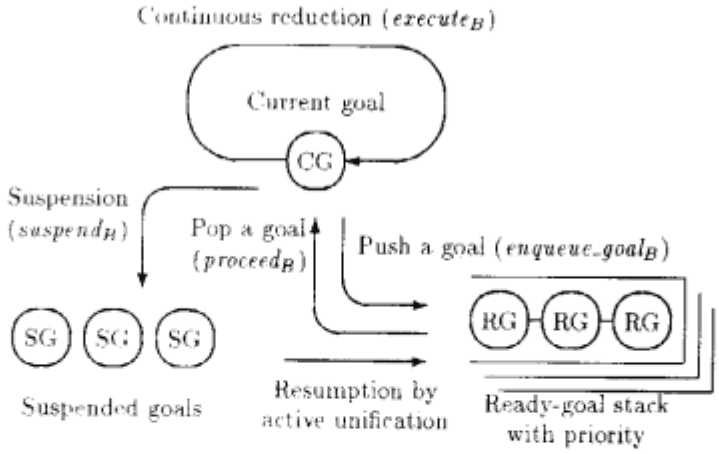

Figure 3: Goal State Transition and KL1-B Instructions

from the ready-goal-stack, then the goal reduction is performed by the KL1-B code corresponding to the goal predicate.

When any unification suspends, the goal is linked as a suspended goal from the variable which caused the suspension [14, 22]. Here, the *non-busy waiting* method has been adopted. That is, the suspended goal is not scheduled until the variable will be instantiated. When a suspended goal is resumed, it is linked to the ready goal-stack again.

Depth-first scheduling is, in principle, adopted for body goals. A left-most body goal can be executed without pushing it to the ready-goal-stack (see figure 3), while other body goals are linked to the ready-goal-stack.

The priority of goal scheduling can be controlled by specifying pragmas [25]. While each shōen is created with the maximum and minimum priority (see section 2.2), the pragmas can specify the relative priority within the range allowed for the shoen. The ready-goal-stack is managed with the priority of goals. The forked goal specified with priority is linked to the specified position. Otherwise, the same priority as with the current goal is adopted.

3.3 Goal Distribution within a Cluster

How to keep the processing load well-balanced is a key issue in making the best use of parallel processing resources. Currently, the following strategies are provided in the KL1 implementation on PIM.

In a cluster, we provided an individual ready-goal-stack with priority to avoid conflicts of access to the common goal-stack [22]. New ready goals with higher priority than the current highest priority are possibly born in a cluster, or sent from other clusters. These higher priority goals are distributed gradually to keep the processor loads in good balance. We found *on-demand* distribution to be an effective way to realize a good balance within a cluster while reducing the amount of wasteful communication among processors [21]. In the on-demand scheme, an idle processor, or a processor executing low priority goals, sends a request to a busy processor executing higher priority goals. On receiving the request, the busy processor sends the goal from its ready-goal-stack to the idle processor. This communication should be done efficiently within a cluster, so we designed a coherent cache and an inter-processor signaling by *slit-checking* for the PIM pilot machines. (See section 8.2.)

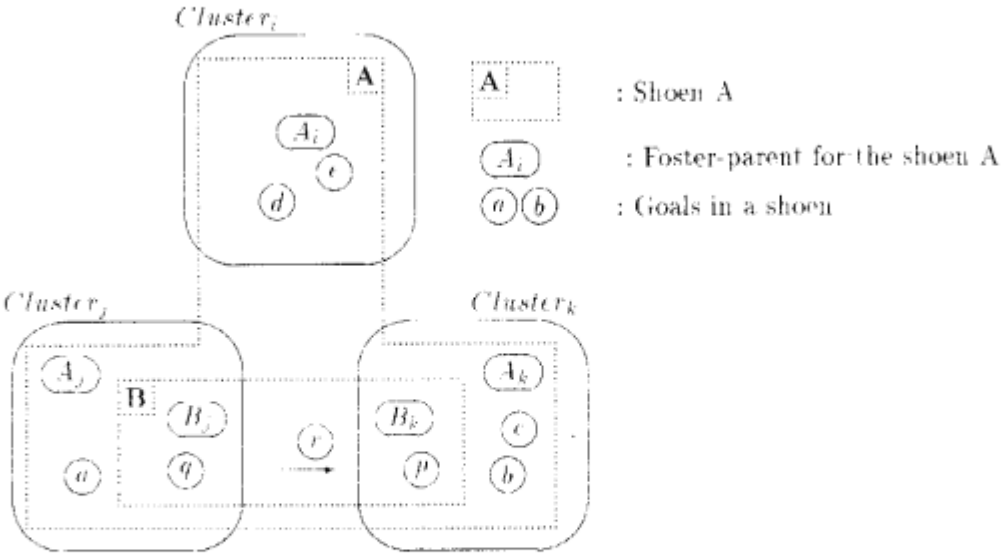

Figure 4: Shoen and Foster-parents

4 Distributed Goal Management

4.1 Inter-cluster Goal Distribution

The load among clusters should be distributed carefully because the communication cost is more expensive than within a cluster. Therefore, we provided pragmas by which users can indicate load distribution while we plan to implement a dynamic load-balancing mechanism.

The pragmas for load distribution are of the form *goal@node(CL)*, attached to body goals as suffixes, and throw KL1 goals to a certain cluster. A body goal *goal@node(CL)* is thrown by a message *%throw* to a cluster *CL* when the clause containing the body goal is committed to. The semantics of programs with pragmas is the same as that without them. The node (more precisely, a certain processing element in the cluster *CL*) that received the *%throw* message links the goal to its ready-goal-stack as well as to the foster-parent [2] as described in the next section.

4.2 Shōen and Foster-Parent Scheme

The main role of a shoen is to control the execution under the shōen, i.e., the shoen status is checked in each goal reduction. Within a cluster, processing elements can share the shōen status, so that the hardware mechanism (a coherent cache, see section 8.4) can reduce the cost of checking the shōen status in every goal reduction. In inter-cluster parallel processing, the shōen tree crosses memory space boundaries of clusters. If we simply represented a link of a shōen tree using an external reference link, the rate of inter-cluster operations would be very high and the synchronization would be very complicated. We provided a *shōen and foster-parent scheme* to avoid this [11].

In the shōen and foster-parent scheme, a foster-parent for a certain shōen is created, if necessary, in a cluster. The foster-parent works as a branch of the shōen within the cluster. The foster-parent manages the child shoens or goals belonging to the shoen in that cluster, i.e., it may start, stop and abort its children. By this scheme, most communication between the child shōens or goals and the parent shōen can be performed by the communication between the children and the foster-parent within a cluster, so that the inter-cluster communication traffic can be reduced.

[2] If there is no foster-parent, one will be created on the spot.

Figure 4 shows the following situation. A shōen A has a child shōen B and several child goals in clusters, $Cluster_i$, $Cluster_j$ and $Cluster_k$. Therefore, each cluster includes a foster-parent (A_i,A_j or A_k). Shōen B has its child goals, p, q and r. They were created at $Cluster_j$ and were linked to the foster-parent B_j. When one goal p is thrown to another cluster, $Cluster_k$, a new foster-parent, B_k, is created, and the goal p is linked to it.

4.3 Weighted Throw Count

Termination detection of some or all processes is one of the principal functions in any system. The end of a KL1 program execution corresponds to the end of the shōen. When all goals in a shōen or descendant shoens are reduced to null, the execution of the shōen finishes.

When all goals under a foster-parent have been reduced to null, the foster-parent sends a termination message to the shōen and disappears. The shoen seems to be able to detect the termination when it receives termination messages from all foster-parents. However, there may be goals in transit as the goal r in Figure 4.

The weighted throw count (WTC) method was provided to solve this problem [20], where a certain weight is assigned for the shōen, its foster-parents, and messages. The WTC can be seen as an application of weighted reference counting [2, 30].

In the WTC scheme, a shōen has a certain weight of negative value, and all its foster-parents and messages have a positive weight. The following condition is maintained during their execution:

$$W_{shoen} + \sum(W_{foster\,parent}) + \sum(W_{message}) = 0$$

For example, when a foster-parent sends a goal to another foster-parent, the sender assigns a certain weight from its own to the goal, then sends the goal with the weight. The receiver adds the weight sent with the goal into its own weight. When a foster-parent disappears, it sends a termination message to the shōen with its weight. When the weight of the shōen becomes zero by adding the weight of the message, the termination of all goals in the shōen is detectable.

5 Distributed Unification

5.1 Export and Import Tables

A goal is thrown by the *%throw* message between the clusters. The *%throw* message includes the following encoded information: the code of the predicate of the goal, the arguments of the goal, and the shoen to which the goal belongs. The encoding of arguments (or any KL1 data) is called *export*; decoding is called *import*.

In the KL1 parallel implementation, an external reference, i.e., a reference to non-local data, is identified by the pair $\langle node, ent \rangle$, where *node* is the cluster number in which the referenced data resides, and *ent* is the unique data identifier in that cluster. We did not choose to take the memory location directly as the identifier, *ent*, because that would make it very difficult to perform garbage collection locally within one cluster. If the locations of data have moved as the result of marking or moving garbage collection (see section 6.2), it must be announced to all clusters that may reference the data. Instead, each cluster maintains an *export table* to register all locations that are referenced from other clusters [14]. Each externally referenced cell is pointed to by an entry in the table, and the entry number is used as the unique identification number. When externally referenced cells are moved as the result of a local garbage collection, the pointers from the export table entries are updated to reflect the movements.

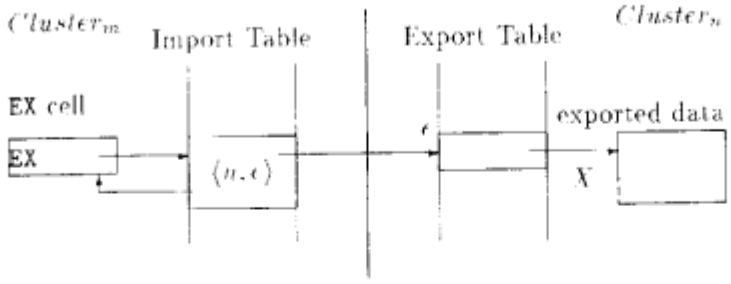

Figure 5: Export Table and Import Table

Also, each cluster maintains an *import table* to register all imported external references. All references in a cluster to the same external reference are represented by internal references to the same *external reference cell*. The external reference cell points to the import table entry and vice versa. Export and import tables are shown in Figure 5, where an external reference cell is indicated by the EX cell[3].

5.2 Avoiding Duplicated Export/Import

Data objects in a cluster may be exported more than once. In such cases, each export tends to use export table entries. In addition, if a cluster imports the same data structure more than once, the cluster must allocate its memory for the same data structure. To avoid the duplicated exports and imports, a hash table is attached to the export table. If a data object is exported more than once, the same export table entry can be retrieved from the object address and used in the second and later export. There is also a hashing mechanism for retrieving an import table entry from an external reference, so that even if a cluster imports the same external reference more than once, only one external reference cell is allocated.

The introduction of export and import tables help reduce the number of inter-cluster read requests as follows. Suppose that $Cluster_n$ exports the same data X twice to $Cluster_m$ as an argument to goals p and q. Since X is exported with the same external reference in the two exports (using the export table mechanism with hashing), $Cluster_m$ allocates only one external reference cell to X (using the import table mechanism with hashing). Even if both p and q attempt to read X, only one read request message is sent to $Cluster_n$, because the first read attempt is remembered by the external reference cell and the second attempt only waits for the return of the value. This mechanism also prevents $Cluster_m$ from making duplicate copies of the same external data.

5.3 Unification Messages

In passive unification, the two terms to be unified are read and compared. To read an external reference (EX) cell to X, a read request is made by sending a message:

$$\%read(X, ReturnAddress)$$

to the referenced cluster, where X is the external reference $\langle n, \epsilon \rangle$ in figure 5, and *ReturnAddress* is new export table entry $\langle m, i \rangle$ for returning the value[4].

If the referenced cell has a concrete value V, it is returned by the *%answer_value* message:

$$\%answer\_value(ReturnAddress, V)$$

[3] EX cell is either an EXREF cell or an EXVAL cell. The data referenced by an EXVAL cell is known to have a concrete value.

[4] The *%read* and *%answer_value* messages correspond to the *%read_value* and *%return value* messages in [14].

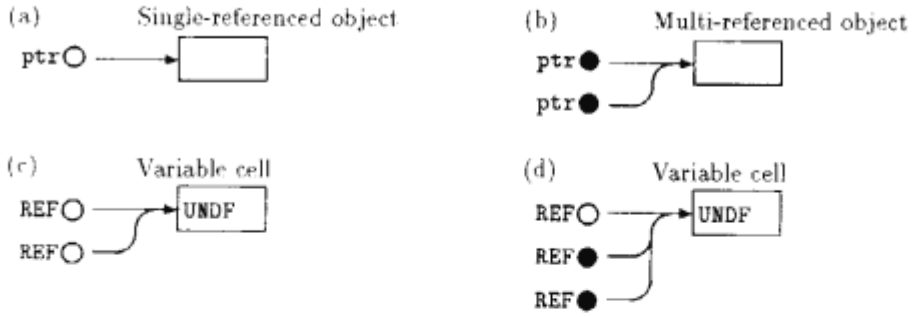

Figure 6: References in the MRB Scheme

If the referenced cell is an unbound variable, the read request is suspended until the variable is instantiated. If it is an EX cell, a *%read* message is passed to the cluster that it references. When the *%answer_value* message returns, the EX cell identified by *ReturnAddress* is overwritten by the value, and the import table entry corresponding to the EX cell can be freed. This is why the cell and the entry are separate.

When an active unification tries to unify an external reference cell X with a term Y,

$$\%unify(X,\ Y)$$

is sent to the referenced cluster. It is a request to unify the data referenced by X with a term Y. The cluster that receives the above message performs the active unification after translating the two terms into internal representations. Care must be taken with the unifications between two unbound variables in different clusters, because they may make reference loops between clusters. This problem can be solved: first, compare the two cluster identifiers, then make sure that the reference pointers point in the same direction, in descending (or ascending) order of cluster identifiers [15].

6 Memory Management in PIM

6.1 Incremental Garbage Collection by MRB

While KL1 can describe synchronization and communication between parallel processes without side-effects, naive implementations of KL1 as well as other concurrent logic programming languages [7, 24, 23] consume memory area very rapidly. For example, whole array elements must simply be copied when only one element is updated because destructive assignment is not allowed. As a result, garbage collection (GC) occurs very frequently. In addition, the locality of memory references is not good during GC by widely used methods, so that cache misses and memory faults occur often. In sequential Prolog [29], this problem is not very serious because of the backtracking feature. However, since concurrent logic programming languages have no backtracking, an efficient incremental GC method is important in their implementation.

The multiple reference bit (MRB) method [5] was proposed as an incremental GC method for concurrent logic programming languages[5]. The MRB method maintains one-bit information in pointers indicating whether the pointed data object has multiple references to it or not. This multiple reference information makes it possible to reclaim storage areas that are no longer used. By keeping information in the pointers rather than in the objects that are pointed to, no extra memory access is required for reference information maintenance.

[5] Another incremental GC method, called lazy reference counting (LRC) [9], was designed. LRC uses two-word indirect pointers with a reference counter.

Figure 6 shows the data representation in the MRB scheme. A single-referenced object (a) and a multi-referenced object (b) can be distinguished by the MRB flag on pointers, *MRB off* by ○ and *MRB on* by ● . Because of the single assignment nature of KL1, an unbound variable cell usually has one reference path for instantiating and one or more reference paths for referencing its value. Therefore, an unbound variable cell with only two reference paths is pointed by MRB off, as in Figure 6(c). On the other hand, an unbound variable with more than two reference paths has only one or no pointer with MRB off, as in Figure 6(d).

The MRB information on variables or structure pointers is maintained through their unification. When a unification consumes a reference path to a single-referenced data object, the storage area can be reclaimed after the unification. For example, the goal reduction by a clause:

$$p([X|Y]) :- true \mid q(X, Y).$$

is committed when the argument of the goal p is the pointer to a cons cell. Its elements are retrieved as the arguments X and Y of the body goal q, consuming one reference path to the cons cell. If the pointer to the cons cell shows *MRB off*, the storage area for the cons cell can be reclaimed during goal reduction.

Although the MRB scheme gives up the storage reclamation for the data objects that were once multi-referenced, the MRB scheme can greatly reduce the memory consumption rate with small runtime overhead. The MRB scheme also makes available several optimization techniques, such as destructive array element update without using the method in Barklund and Millroth (1987). In addition, the MRB scheme can be used for the export and import procedures in section 5.2. Because single-referenced data objects may not be exported more than once, we introduce two kinds of export and import tables, one each for single-referenced objects and multi-referenced objects. While the export and import tables with the hashing function are used for multi-referenced data objects that can be found by the MRB scheme in each cluster, a simpler external reference mechanism is used for single-referenced objects.

6.2 Garbage Collection within a Cluster

Data structures or variables in KL1 are stored as shared data in each cluster memory. The MRB scheme enables storage reclamation for these data structures. Thus, free lists for data structures and variable cells are maintained. Storage allocation and reclamation are very frequent operations. So each processing element has a set of free lists for frequently used cells, enabling each free list access to be done independently in each processing element.

We use another type of garbage collection that is performed locally within a cluster accompanied with the incremental garbage collection by MRB. This is because the MRB scheme leaves some garbage. We first implemented a simple garbage collection, the *copying* scheme, on our experimental KL1 system.

We designed a parallel mechanism that enables all processing elements to collect garbage in a cluster. When a certain processing element has a shortage of memory space during its goal reduction, it reports this event to other processing elements after it finishes the current goal reduction. This is because garbage collection is difficult to start during goal reduction. Shortage of memory space should therefore be detected before all the memory area is used up. After all processing elements stop their goal reductions, they start the copying operations, tracing all active cells in the shared memory of a cluster. Here, the copying roots are the ready goals in ready-goal-stacks[6].

[6]The export tables in section 5 are also the roots of copying operations.

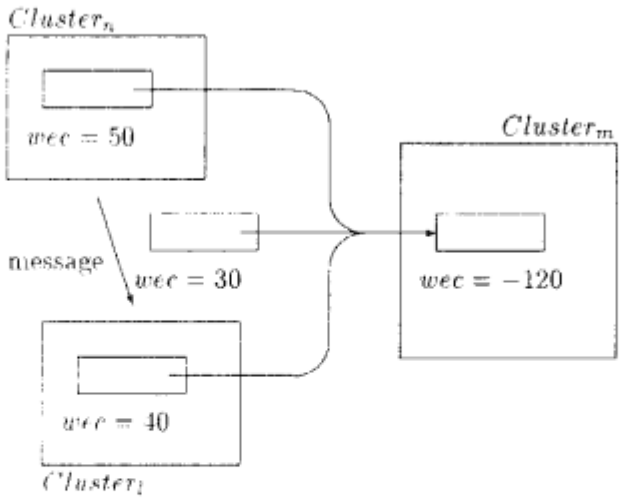

Figure 7: WEC Scheme

6.3 Distributed Garbage Collection by WEC

Since export table entries for multi-referenced data objects cannot be freed by local garbage collection within a cluster of the type described in section 6.2, there must be an inter-cluster garbage collection mechanism to free those entries that have become garbage.

The weighted export count (WEC) scheme was designed as incremental inter-cluster garbage collection. The WEC scheme can also be seen an application of weighted reference counting [2, 30]. Note that a naive implementation of the standard reference counting scheme does not work correctly in a distributed environment.

The WEC scheme assigns weighted export counts (wec) to references (pointers) as well as to referenced data [15]. More precisely, positive values are assigned to external references (import table entries and references encoded in messages), and negative values are assigned to export table entries, so that the following condition is kept for every export table entry E (see figure 7.):

$$(weight\ of\ E) + \sum_{x: reference\ to\ E} (weight\ of\ x) = 0$$

The weight of E will become zero only when there is no reference to E. As a result, export table entries can be incrementally reclaimed through the message operation with wec.

7 Abstract Instruction Set: KL1-B

To build an efficient parallel inference machine, execution on each processing element must be as efficient as possible. Therefore, KL1-B was designed first based on sequential execution[7]. It was extended for parallel execution.

Most instructions in KL1-B include run time data type checks. The actions that follow the run-time type check are very different. Therefore, all the memory words and all the argument/temporary registers hold tagged words of the form:

$$\langle tag(MRB, Type), value \rangle.$$

The MRB in each tag is maintained to show the multiple reference information. $Type$ shows the data type information.

[7] An explanation of each KL1-B instruction can be found in [5] and [16].

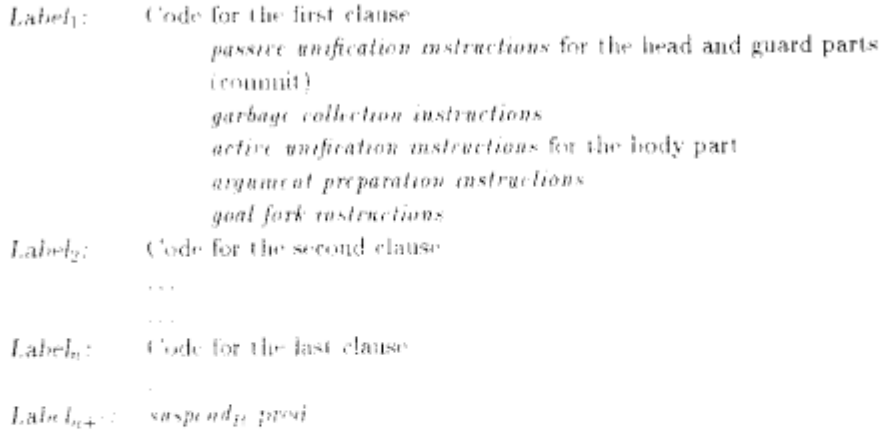

Figure 8: Form of Compiled Codes in KL1-B

7.1 Compiled Code in KL1-B

A data structure called a *goal-record* is used for representing a goal. A goal-record consists of its argument list, a pointer to the compiled code corresponding to its predicate name, and control information. The argument list includes atomic values or pointers to variables or structure bodies in the heap.

A goal reduction is initiated by a KL1-B instruction[8], $proceed_B$, popping a goal as a current goal from the ready-goal-stack. Here, we assume that the arguments of a current goal are located in argument registers (Ais).

A set of candidate clauses for a predicate is compiled into a sequence of KL1-B instructions[9] as shown in figure 8. For the current goal, candidate clauses are tested sequentially by head unification and guard execution to choose one clause whose body goals will be executed.

A KL1-B code for a set of candidate clauses includes passive unification instructions for head and guard parts, active unification instructions, argument preparation instructions and goal fork instruction for body part, and garbage collection instructions. The guard part is compiled so that argument registers are never destroyed before commitment. Instructions are arranged so that reference paths to data objects can be maintained correctly in terms of the MRB scheme. Table 1 shows the principal KL1-B instructions.

7.2 Passive Unification

Passive unification instructions include the instructions for goal arguments ($wait\_XXX_B$), and for structure elements ($read\_XXX_B$). The indexing instructions are also used to avoid duplicated operations between the head and guard part execution of candidate clauses.

Dereferencing is required at the beginning of passive and active unification instructions. The data type of an argument register is first tested to see whether its content is an indirect pointer or not. If it is an indirect pointer, the pointed cell is dereferenced until an instantiated value, an unbound variable cell, or an external reference is reached.

If the instantiation of a variable (including an external reference) is required during the execution of the passive part, the test for this clause is abandoned. The variable that caused the suspension is saved in a suspension stack, then execution proceeds to the next candidate clause.

[8] In this article, each KL1-B instruction is written with postfix B, for example, $proceed_B$.

[9] The actual compiled code has a different form when indexing instructions are used.

Table 1: Principal KL1-B Instructions

KL1 B Instruction	Comment
Passive unification and suspension	
wait_value$_B$ *Ai, Aj*	Wait for two instantiated terms, *Ai* and *Aj*, then unify them.
wait_const$_B$ *Ai, C*	Wait for a constant value *C*.
wait_list$_B$ *Ai*	Wait for a pointer to the list.
read_car/cdr_var$_B$ *Ai, Aj*	Read the car (or cdr) of a list cell pointed by *Ai* into *Aj*.
try_me_else$_B$ *Label*	Set a branch label *Label*.
suspend$_B$ *Goal*	Suspend *Goal*.
MRB maintenance and garbage collection	
mark$_B$ *Ai*	Set the MRB of *Ai* to on.
collect_value$_B$ *Ai*	Reclaim along with the reference path from *Ai*.
collect_list$_B$ *Ai*	Reclaim a list cell *Ai*.
Active unification	
get_value$_B$ *Ai, Aj*	Actively unify *Ai* with *Aj*.
get_const$_B$ *Ai, C*	Unify *Ai* with a constant *C*.
get_list_value$_B$ *Ai, Aj*	Unify *Ai* with a list pointed to by *Aj*.
Argument preparation	
put_var$_B$ *Ai, Aj* ; *set_value*$_B$ *Gi, Aj*	Make a new variable pointing from *Ai/Gi* and *Aj*.
put_value$_B$ *Ai, Aj* ; *set_value*$_B$ *Gi, Aj*	Move the variable from *Aj* to *Ai/Gi*.
put_const$_B$ *Ai, C* ; *set_const*$_B$ *Gi, C*	Put a constant *C* in *Ai/Gi*.
put_list$_B$ *Ai* ; *set_list*$_B$ *Gi*	Allocate a list cell in *Ai/Gi*.
write_car/cdr_var$_B$ *Ai, Aj*	Make a new variable pointed to by *Aj* and the car (or cdr) of a list Ai.
write_car/cdr_value$_B$ *Ai, Aj*	Move *Aj* to car (or cdr) of a list *Ai*.
write_car/cdr_const$_B$ *Ai, C*	Write a constant *C* in car (cdr) of a list *Ai*.
Goal fork	
proceed$_B$	Pop a goal from the ready-goal-stack.
execute$_B$ *Goal*	Jump to the code for *Goal*.
enqueue_goal$_B$ *Goal*	Push a forked goal *Goal* to the ready-goal-stack.
enqueue_with_priority$_B$ *Goal, Pri*	Push *Goal* to the ready-goal-stack of the priority *Pri*.
enqueue_to_processor$_B$ *Goal, Node*	Throw *Goal* to *Node*.

Note: *Ai, Aj*: Argument registers *Gi*: *Gi*-th argument of a forked goal
C: Atoms, integers and nil are handled by individual instructions.

```
put_list_B          Aj          % allocate a cons cell
write_car_const_B   Aj, foo     % write the car part
write_cdr_variable_B Aj, Ak     % allocate a new variable
get_list_value_B    Ai, Aj      % active unification
```

Figure 9: An Active Unification Example

The *%read* message is not sent, in principle, in passive unification instructions even when the value of a certain external reference cell is required; instead, such a message will be sent in the $suspend_B$ instruction, because other candidate clauses may be committed.

7.3 Suspension

If no clause is selected for the current goal, the $suspend_B$ instruction tests the suspension stack. If there is no variable, a failure exception occurs at the shoen. Otherwise, the current goal becomes a suspended goal. First, variables that cause the suspension are popped up from the suspension stack. Then, the current goal is linked to these variables, setting the tag of the variable by `HOOK`, to realize a non busy waiting synchronization mechanism between KL1 goals.

When an external reference is found in the suspension stack, the *%read* message is sent to the node where the exported data resides (see figure 5). The goal waits for the *%answer_value* message as a suspended goal.

The processing element that received *%read* message returns the value of the exported data with the *%answer_value* message. However, the exported data may be an unbound variable cell. In this case, the action of replying to the *%read* message is suspended by linking a *reply_record* to the unbound variable cell. The reply record can be seen as a special goal record to reply with the *%answer_value* message.

7.4 MRB Maintenance and Garbage Collection

Active unification may produce a chain of variable cells pointed to by indirect pointers. These variable cells pointed to by an indirect pointer with *MRB off* can be reclaimed during deferencing. Therefore, each dereferencing operation includes the MRB test and, possibly, a reclamation operation.

The MRB is maintained in each KL1-B instruction. In addition, several garbage collection instructions are introduced in KL1-B. The compiler detects candidate places where reference paths are added. $mark_B$ is used to set MRB to *on*. When the compiler finds a unification in which a reference path to a data object is consumed, it inserts a $collect\_XXX_B$ instruction at an appropriate place. $Collect\_list_B$ is a typical KL1-B instruction which corresponds to the goal reduction by the clause in section 6.1. Note that the cons cell can be reclaimed after the clause is committed. Therefore, $collect\_list_B$, which reclaims the cons cell if it is a single-referenced cell (*MRB off*), is put after the passive unification instructions as shown in figure 8.

7.5 Active Unification and Resumption

If a clause is selected, the body part of that clause is executed. Execution of the body part includes two kinds of operations, *active unification* and *body goal fork*. Figure 9 shows the typical compiled code for the active unification in such a clause as:

$$\ldots \mid X = [foo|Y], \ldots$$

The structures for active unifications or the arguments for body goals are prepared by argument preparation instructions, put_XXX_B, $write\_XXX_B$, and set_XXX_B. New variable cells or structures,

such as the right-hand side of the above unification, may be allocated from free lists or in free memory area by these instructions. Unlike the original WAM, structure elements should not be used directly as undefined variable cells, to avoid fragmentation. This is because the incremental garbage collection by MRB may reclaim a structure body and its elements at a different timing. Thus, when a structure element should be initiated as a new variable, the new variable cell is allocated separately from the structure body, and a pointer to the cell is stored inside the body.

The last instruction in figure 9, *get_list_value*$_B$, is a typical KL1-B instruction for active unification. This instruction has one of four kinds of actions, selected by checking the data type. When *Ai* is an uninstantiated variable without suspended goals, *Aj* (a pointer to a cons cell made by the first instruction in Figure 9) is assigned to the variable cell. Note that unbound variables are located in shared memory. Thus, the instantiation of unbound variables is done by locking and unlocking the variable cells [22]. Here, it is important to shorten the period of locking the unbound variable. Therefore, the compiler generates the compiled code as shown in Figure 9, where the right-hand side structure is created first. As a result, the unbound variable is locked only within the *get_list_value*$_B$ instruction.

If *Ai* is an uninstantiated variable with suspended goals, these suspended goals are resumed by moving the goal-records linked from the variable to the ready-goal-stack again before instantiating to *Aj*. (See figure 3.) When reply_records are linked to that variable, the *%answer_value* messages for each reply_record are sent to the cluster which is waiting for the instantiated value.

Ai may be an external reference. In this case, the *%unify* message is sent to the node which exported the variable. The node which received the *unify* message performs active unification on the behalf of the sender processor.

When *Ai* is a pointer to a list cell, general unification is performed. Otherwise, unification fails and an exception occurs.

7.6 Goal Fork and Slit-checking

Several goal fork instructions are provided to push and pop a goal-record to and from a ready-goal-stack, or to execute goal reductions repeatedly. As shown in Figure 3, a KL1 B instruction *proceed*$_B$ pops up a goal record (a current goal) from the ready-goal-stack when the previous goal reduction did not fork any body goals. The KL1-B code corresponding to the goal predicate is executed. Assume that there are two body goals in a KL1 clause as:

$$p :- \langle guard \rangle \mid q, r.$$

The reduction of the left-most body goal, q, will follow just after the current goal reduction, while the other goal(s), r, is pushed to the ready-goal-stack.

Other body goals are pushed by the *enqueue_goal*$_B$ instructions. When scheduling priority is specified by the pragmas, the KL1 compiler generates a KL1-B instruction, *enqueue_with_priority*$_B$. When the pragmas for load distribution are specified in a KL1 program, KL1-B instructions *enqueue_to processor*$_B$ are used. This instruction sends a message, *%throw*, to the specified cluster instead of enqueuing its own ready-goal-stack.

The following events are incidental in KL1 execution: a garbage collection requirement (section 6.2), an inter-processor communication request, and a goal fork with the highest priority (section 3.3). These events are only detected by *slit-checking* in the *execute*$_B$, *proceed*$_B$ and *suspend*$_B$ instructions, i.e., the actions corresponding to these events are delayed until a certain goal reduction finishes, even if the event occurred during a goal reduction. This is because garbage collection is difficult to start during a goal reduction. In inter-processor communication or for a goal fork with

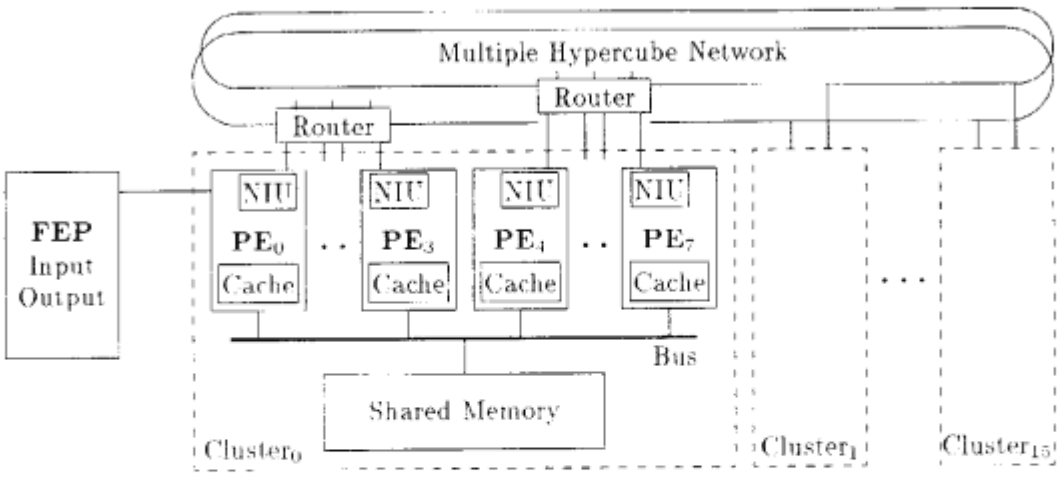

Figure 10: The Pilot Machine: PIM/p

highest priority, the corresponding actions do not have to be performed immediately, so they may be delayed until after the goal reduction finishes.

As described in section 4.2, a foster-parent in a cluster holds the shōen status as well as information about the computing resources assigned for the foster-parent. Before $execute_B$, $proceed_B$ or $suspend_B$ starts a goal reduction, it checks the shōen status of the current goal, and the computing resources left in that foster-parent.

8 PIM Hardware Architecture

8.1 Targets of the PIM Hardware Architecture

Our performance target in the R&D of PIM hardware architecture was to execute KL1 programs with more than 100 times the performance of conventional machines. To achieve this goal, we studied new processing element architectures as well as new parallel architectures to connect more than 100 processing elements. The target processing element performance is 200K to 500K RPS[10], so that 10 to 20M RPS is expected to be the total performance for practical applications.

Several pilot machines are now being developed for the PIM research for the final stage of the FGCS project. The PIM/p is one of the PIM pilot machines, which is planned to have 128 processing elements. The rest of this section focuses on the hardware architecture of the PIM/p.

8.2 The Pilot Machine: PIM/p

In the parallel architecture design for the PIM/p, we aim to build a parallel processing architecture where the locality in communication cost can easily be used from software. We introduced a hierarchical structure, as shown in Figure 10. Eight processing elements (PEs) form a cluster with shared memory. The PIM/p consists of 16 clusters connected by inter-cluster network.

8.3 PIM/p Processing Element

A PIM/p processing element is implemented on a single board with about 20 static RAMs and several custom CMOS LSIs, a CPU, a network interface unit (NIU), cache controller units (CCUs), and a floating point processor unit (FPU), as shown in Figure 11. The basic machine cycle target is 50

[10] RPS: KL1 goal reductions per second

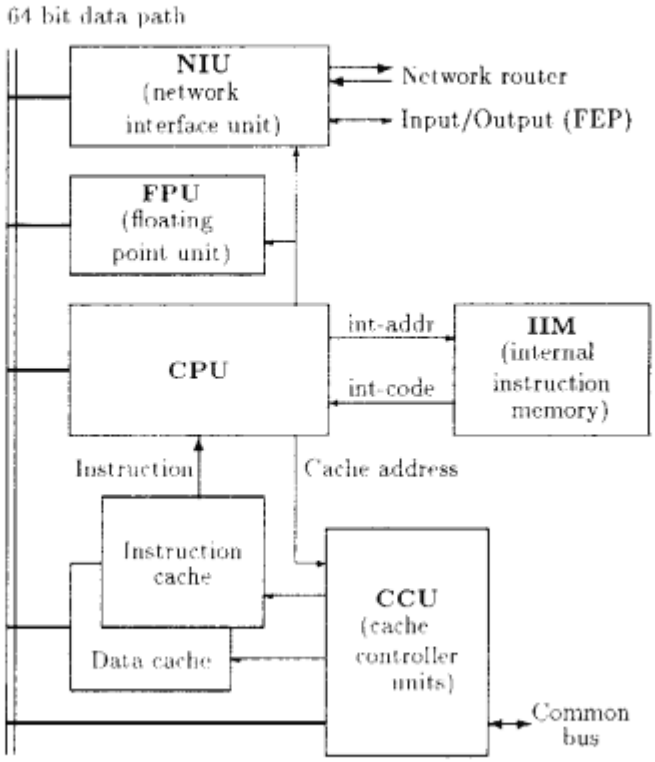

Figure 11: PIM/p Processing Element Configuration

nanoseconds.

The processing element includes two caches: an instruction cache and a data cache. The contents of both cache memories are identical. They are provided to enable the CPU to fetch both data and instructions every machine cycle. The cache controller units (CCUs) manage both the instruction cache and the data cache. The cache address array is updated by both commands from the CPU and a common bus.

The CPU has two instruction streams: one from the instruction cache, called external instructions, and the other from the internal instruction memory (IIM), called internal instructions. *External instructions* are used to represent compiled codes of user programs. *Internal instructions* are stored in the internal instruction memory (IIM) of each processor, in the same way as in the microprogrammable processor. Small programs in IIM can specify the complex actions of KL1-B instructions. Both instructions include KL1 support instructions as well as simple RISC-like instructions. They are invoked by external macro-call instructions. Hopefully, the CPU will execute an instruction every 50 nanoseconds using a four-stage pipeline in most cases.

Slit-check and Interrupt

A hardware mechanism for *slit-checking* (see section 7.6) is incorporated into the processing element of PIM/p. A normal hardware interrupt causes automatic save of program status: however, slit-checking does not. Each processing element has a dedicated register, each bit of which can keep an individual event. The slit-checking mechanism has an additional one-bit flag to show whether any events happened or not, which can be tested by one conditional branch instruction. On general-purpose computers, slit-checking might be implemented using normal interrupt mask/unmask operations and a cumbersome interrupt handler, which would incur too great a cost for the KL1 system. By incorpo-

Table 2: Basic CPU Commands to the Cache

CPU command	*Comment*
Read	Ordinary memory read.
Write	Ordinary memory write.
Read_Invalidate	When cache-to-cache transfer occurs, the source cache block is invalidated. Otherwise, same as Read.
Read_Purge	After CPU reads, the cache block is purged. The shared blocks in other caches are also purged.
Direct_Write	If the cache misses at the block boundary, write data to the cache without fetching from memory. Otherwise, ordinary memory write.
Lock_Read	Lock address, then memory read.
Write_Unlock	Memory write, followed by unlock.
Unlock	Unlock address.

rating the hardware slit-checking mechanism, the processing element can avoid frequent mask/unmask operations and interrupt handling overhead.

Registers

The processing element includes 32 general-purpose registers with some dedicated registers. Each general-purpose register has an 8-bit tag and 32-bit data. The dedicated registers include a condition code register for the result of ALU execution and a slit-check register. Most flags, such as the condition code, are placed in the tag part of the dedicated registers. Therefore, these flags can be tested by the tag branch instructions.

A CPU has virtual registers, called indirect registers, in addition to the above registers. Through the indirect registers, internal instructions can easily handle the operands of a macro-call instruction that has just invoked the internal program code. In other words, each indirect register corresponds to the operand position of the macro-call instruction. It can represent either the immediate value or the contents of a register specified in the operand of the macro-call instruction.

8.4 Cache System

Processing elements within each cluster share one address space. Therefore the design of a local coherent cache is a key issue in increasing the efficiency of local execution on each processing element, and it enables high speed communication within a cluster. Although several coherent cache protocols have been proposed so far [1, 8, 3, 19], we designed a cache protocol for KL1 parallel execution based on the simulation results [10]. The simulation results have shown that KL1 programs require more write accesses than conventional languages. Therefore, we chose a write-back protocol which can reduce common bus traffic more than a write-through protocol. When a cache block is updated, the consistency with other caches is kept by invalidating the shared cache blocks in other caches. In addition, we extended some cache functions from ordinary cache protocols using the characteristics of the KL1 parallel execution. Table 2 shows the basic CPU commands to the cache.

Cache Commands for KL1 Support

In parallel implementation of KL1, some data structures can be found out when they are not accessible. A typical example is an explicit communication between processing elements. First, a sender processor creates a message in its own cache. The message is sent to a receiver processor as a *cache-to-cache* data transfer. Although the message in the sender processor is useless after message transfer, it remains as

shared cache blocks between both processors' caches. Therefore, when the receiver processor makes a message in the same area, another cache will be invalidated. The CPU command, *Read_Invalidate*, is provided to avoid invalidation caused by invalidating at cache-to-cache data transfer.

In normal write operations, *fetch-on-write* is used. However, when data structures are created in an unused memory area, it may not be necessary to fetch-on write. This is because the memory contents have no meaning, and because the new data structure is not shared by other processors. The *Direct Write* command is introduced to avoid useless cache block fetch from shared memory. The *Read_Purge* command invalidates its own cache block just after the CPU reads the last cache block word, so that the *Direct_Write* command can be used for already-used memory area that is already in use.

Hardware Lock

Lock operations are essential for implementing KL1 in the shared memory multiprocessor. This is because exclusive memory access is required to instantiate variables in active unifications (see section 7.5) or to link suspended goals to them (see section 7.2). Although lock conflicts seldom occur, lock latency is high in KL1 execution. The simulation results in Matsumoto et al. (1987) show that the *Read_Lock* frequency is about 7 % for data access, so a *lightweight* lock operation is required.

The PIM/p cache enables a *lightweight* lock and unlock operation by using the cache block status, lock address registers, and busy-wait locking scheme. When the CCU receives a *Lock_Read* command from the CPU, the CCU checks the corresponding address tag and status tag. If the address hits and its status is *exclusive*, the address can be locked without using the common bus. The locked address is held in a lock address register.

8.5 Hyper-Cube Network and Network Interface Unit

As discussed in sections 5 and 7, inter-cluster communication may be required during a KL1-B unification instruction on each processing element. That communication may include various kinds of messages. We designed the inter-cluster network aiming at enough performance for both short and long message packets, and inter-cluster processing where it is required. The hyper-cube structure [4] has been introduced to connect clusters in PIM/p, placing each cluster on the hyper-cube node. This is because the hyper-cube structure enables us to shorten the inter-cluster distance with reasonable hardware costs. In addition, the network router can be implemented distributedly on each cluster.

The network was designed aiming at an inter-cluster communication throughput of 40 M byte/s. We chose the following configuration considering the limitations in hardware implementations. A network router was designed for a six-dimensional hyper-cube connection. While four dimensions are enough to connect 128 processing elements (16 clusters), the router switch will be available for future extensions. Each communication path has a throughput of 20 M byte/s one byte every 50 nanoseconds, in both directions. To obtain 40 M byte/s throughput, the inter-cluster network has been doubled. Therefore, two network routers are provided for each cluster, one for four processing elements.

Each processing element has a network interface unit (NIU) as a co-processor of the CPU. The NIU has two packet buffers, one for each direction, whose contents can be transferred to and from CPU registers. A packet is sent to the other processing element from the NIU by the CPU requests. The buffer status in a NIU, full or empty, can be reported to the CPU by the slit-checking mechanism. Therefore, these message handling operations can be done on each processing element.

8.6 PIM/p Instructions

The instruction set for the PIM/p processing element has been designed for the efficient implementation of KL1. The design started by analyzing the behavior of the KL1-B instructions [26].

Tagged architecture

As discussed in section 7, run-time data type checks are essential for KL1-B instructions, so we introduced the tagged-architecture in the CPU design. The tag part in a KL1 variable cell can be implicitly loaded and stored with the data part by using basic memory access instructions. In addition, a new tag can be given in memory access instructions and ALU computation. The memory access giving a new tag is a primitive operation in the KL1-B argument preparation instructions of KL1-B.

The run-time test of the type tag is a primitive operation to implement KL1. Most unification includes a multi-way branch for the goal argument type. Some Prolog machines, such as the PSI [18], have a hardware-supported multi-way branch function. However, the processing element of PIM/p does not have such hardware. This is because (1) it is difficult to adopt a hardware-supported multi-way branch to a pipeline processor, and (2) branches taken in run-time are biased. Even a normal two-way branch can be useful enough by selecting an appropriate branch condition. Therefore, the PIM/p instruction set has only two-way branch instructions, but various tag conditions can be specified in them. A branch condition can be specified as a logical operation between two register tags, or between a register tag and an immediate tag. In addition, some branch instructions have an immediate tag mask in their operands.

Conditional macro-call instruction

The next issue is how to implement polymorphic functions in KL1 B, because most KL1-B instructions include very different actions that follow the run-time data type check. The RISC like instruction set can be executed using short pipeline cycles, and in hardware design cost is relatively low. However, considering the naive expansion of KL1-B using RISC-like instructions, the static code size of compiled programs will be very large. This problem can be solved by incorporating the features of microprogrammable processors such as PSI [18]. Therefore, we designed RISC-like instructions with conditional macro-call instructions for the PIM/p processing elements, so that the advantages of both in the RISC-like instructions and microprogrammable processors are available in the KL1-B implementation on the PIM/p.

Macro-call instructions were introduced to implement high-level KL1-B instructions. A macro-call instruction can be regarded as a *lightweight* subroutine call or as a high-level instruction realized by the microprogram. A macro-call instruction invoke a small program in the internal instruction memory (IIM) depending on given conditions, which has the form:

MCALL if *cond*, *address* **with** $\mathtt{reg}_0$, $\mathtt{reg}_1/\mathtt{immed}_1$, ..., $\mathtt{reg}_n/\mathtt{immed}_n$

where:

address :	Entry address of the internal instruction memory
$\mathtt{reg}_i/\mathtt{immed}_i$:	register number or constant for the macro-call argument
cond :	condition for the macro-body invocation.

A tag condition, *cond*, can be specified as a logical operation between a register tag, $\mathtt{reg}_0$ and a register tag, $\mathtt{reg}_1$, or an immediate tag, `immed`.

MRB GC support

The principal operations such as incremental garbage collection by MRB and dereferencing are supported by dedicated RISC-like instructions. In MRB incremental garbage collection, each variable cell or structure is allocated from a free list. When reclaimed, its memory area is linked to a free list. To support these free list operations, the `PUSH` and `POP` instructions are used. `PUSH` can link a variable cell or a structure to the free list, and `POP` can allocate it from the free list, in one machine cycle.

The MRB of each pointer and data object must be maintained correctly in all unification instructions. Here, the most primitive operation is MRB maintenance during dereferencing. In dereferencing, the MRB of the dereferenced result should be *off* if and only if MRBs of both the pointer and the cell are *off*. In this case, the indirect word cell can be reclaimed immediately because there are no other reference paths to it. Two dedicated instructions, `MRBorRead` and `DEREF`, support this operation. `MRBorRead` accumulates both the address register's MRB and the destination register's MRB, then sets the result in the destination register. `DEREF` performs MRB accumulation along with the `POP` operation.

Special cache access

As stated in section 8.4, the coherent cache of the processing element has extended functions for KL1 parallel execution. The instruction set includes memory access instructions corresponding to each cache function: `DirectWrite`, `ReadPurge`, `ReadInvalidate`, and `ExclusiveRead`. Exclusive memory access instructions, `LockRead` and `WriteUnlock`, are also provided. Incorrect use of these instructions may cause fatal errors. Therefore, the use of these instructions will be limited to internal instructions.

The processing element performance estimated from the compiled code is over 600 K RPS for the append program. Note that the estimated performance includes the incremental garbage collection cost using MRB.

9 SUMMARY

This paper outlined the parallel inference machine architecture, KL1 parallel implementation issues, such as distributed resource management, goal scheduling and distribution, memory management, and distributed unification, were discussed, based on the logic programming framework. These have been implemented on the parallel software workbench, the Multi-PSI systems. This paper described the design of the PIM pilot machine hardware, including its processing element instruction set. The LSIs are now being implemented.

ACKNOWLEDGEMENT

All the parallel inference machine systems research has been performed with the collaboration of all PIM and Multi PSI researchers in the FGCS project. Most ideas for the KL1 parallel implementations came from the accumulation of their discussions.

I wish to thank all researchers of the companies participating in the PIM R&D project: Fujitsu Limited, Mitsubishi Electric Corporation, Hitachi Ltd., and Oki Electric Industry Co. Ltd. One of the PIM pilot machines, PIM/p, shown in this report was designed and implemented by cooperative work with Mr. A. Hattori, Mr. T. Shinogi, Mr. K. Kumon and all of their colleagues at Fujitsu Limited.

Finally, I would like to thank the ICOT Director, Dr. K. Fuchi, and the chief of the fourth laboratory, Dr. S. Uchida, for their valuable suggestions and guidance.

References

[1] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transaction of Computer Systems*, 4(4):273–298, 1986.

[2] D.I. Bevan. Distributed Garbage Collection using Reference Counting. In *Proceedings of Parallel Architectures and Languages Europe*, pages 176–187, June 1987.

[3] P. Bitar and A. M. Despain. Multiprocessor cache synchronization. In *Proc. of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[4] G. Broomell and J.R. Heath. Classification categories and historical development of circuit switching topologies. *ACM Computing Surveys*, 15(2):95–133, 1983.

[5] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 276–293, 1987.

[6] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.

[7] K. Clark and S. Gregory. Notes on Systems Programming in PARLOG. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 299–306, Tokyo, 1984.

[8] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proc. of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, 1983.

[9] A. Goto et al. Lazy Reference Counting: An Incremental Garbage Collection Method for Parallel Inference Machines. In *Proc. of the Joint Fifth International Logic Programming Conference and Fifth Logic Programming Symposium*, pages 1241–1256, Seattle, WA, August 1988.

[10] A. Goto, A. Matsumoto, and E. Tick. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *16th Annual International Symposium on Computer Architecture*, Jerusalem, May 1989.

[11] A. Goto et al. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, Japan, November 1988.

[12] A. Goto and S. Uchida. Current Research Status of PIM: Parallel Inference Machine. TM 140, ICOT, 1985. (Third Japan-Sweden workshop on Logic Programming, Tokyo).

[13] A. Goto and S. Uchida. Toward a High Performance Parallel Inference Machine -the Intermediate Stage Plan of PIM–. In *Future Parallel Computers*, pages 299–320, LNCS 272, Springer Verlag, 1986.

[14] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[15] N. Ichiyoshi et al. A New External Reference Management and Distributed Unification for KL1. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.

[16] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 468–477, 1987.

[17] K. Nakajima et al. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, Lisboa, June 1989.

[18] H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine: PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, pages 104–113, San Francisco, 1987.

[19] M.S. Papamarcos and J.H. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, 1984.

[20] K. Rokusawa et al. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 1 Architecture, pages 18–22, August 1988.

[21] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *Proceedings of IFIP Working Conference on Parallel Processing*, Pisa, Italy, April 1988.

[22] M. Sato, A. Goto, et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 338–355, 1987.

[23] E.Y. Shapiro, editor. *Guarded Horn Clauses*, pages 140-156. MIT Press, 1987.

[24] E.Y. Shapiro. A subset of Concurrent Prolog and Its Interpreter. TR 003, ICOT, 1983.

[25] E.Y. Shapiro. Systolic programming: A paradigm of parallel processing. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 458-470, 1984.

[26] T. Shinogi et al. Macro-call Instruction for the Efficient KL1 Implementation on PIM. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, Japan, November 1988.

[27] Y. Takeda et al. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.

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

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

[30] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architecture. In *Proceedings of Parallel Architectures and Languages Europe*, pages 432-443, June 1987.