PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

OVERVIEW OF THE PARALLEL INFERENCE MACHINE ARCHITECTURE (PIM)

Atsuhiro Goto Masatoshi Sato Katsuto Nakajima
Kazuo Taki Akira Matsumoto

Institute for New Generation Computer Technology
4-28, Mita-1, Minato-ku, Tokyo 108, Japan

ABSTRACT

As part of the FGCS project, we are developing parallel inference machine (PIM) systems based on a logic programming framework. The PIM systems include the kernel language (KL1), the parallel operating system (PIMOS) and the PIM hardware architectures.

KL1 has been designed with its parallel implementation techniques. We used the characteristics of KL1 to solve the KL1 parallel implementation issues, 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, we aimed at a total effective performance of 10 to 20 Mrps. We introduced a hierarchical configuration to connect more than one hundred processing elements. We provided a new instruction architecture for KL1 for the processing elements. We designed a coherent cache protocol to make high-performance clusters, each of which includes eight processing elements connected with shared memory. We designed a multiple hypercube network to connect these clusters.

1 INTRODUCTION

The research and development (R&D) of the parallel inference machine (PIM) system is one of the most important targets in the FGCS project. The PIM systems will be the pioneer of parallel processing in knowledge information processing system (KIPS) application fields (Murakami et al. 1985).

During the initial stage (1982 to 1984) of the FGCS project, the elementary mechanisms of PIM were studied from various standpoints (Murakami et al. 1985, Goto and Uchida 1985, Goto and Uchida 1986). The R&D of the current PIM system started in 1985. It includes the design and implementation of the kernel language (KL1), the PIM operating system (PIMOS), and the PIM hardware architecture as shown in Figure 1. We

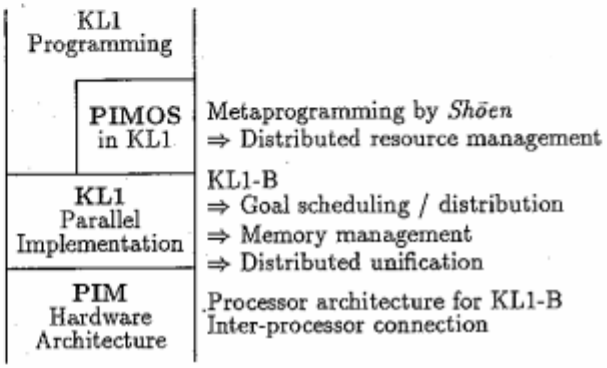

Figure 1: Parallel Inference Machine System Overview

set the following goals for the R&D of the PIM system.

1.1 Research Goals

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 so-called *semantic gap* argument: application and implementation are closer, therefore execution is faster.

KL1, the kernel language of the PIM system, was designed based on GHC (Ueda 1986a). The major reasons for choosing GHC 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, GHC is an efficient language, in the sense that we can specify the machine level language (Goto 1987).

We hope to realize very high execution performance for the logic programming in KL1. We believe that more than one hundred times the performance of current machines will be necessary to enhance the logic program-

ming application research. Parallel machine architecture research to date has explored many new technologies (Hwang and Briggs 1984), but there remain many unsolved problems. To achieve this performance goal, both software and hardware architectures have been studied.

Next, we aimed to build practical systems that would be available as research tools in the next stage of the project. This is essential for the 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 tried to build the PIM system by KL1. The PIM operating system, PIMOS (Chikayama et al. 1988), is written in KL1 as a *self-contained* operating system. In addition, the language features of KL1 are fully used in the parallel architecture design.

1.2 Issues in the Following Sections

This report describes the parallel execution mechanism of KL1 and the hardware architecture of the PIM pilot machine. The major issues are as follows.

Distributed management for KL1 programs: The first issue in the KL1 parallel implementation is how to control the KL1 programs in distributed environments. The meta-programming capability by the *shōen* (Chikayama et al. 1988) facility was introduced to KL1 to manage KL1 programs by PIMOS written also in KL1. The next section describes how to realize the shōen facility: more precisely, the shōen and foster-parent scheme with weighted message protocol is discussed.

Scheduling/distribution: Scheduling and distribution of KL1 goals are the key issues for the efficient implementation of KL1. Section 3 describes the non-busy waiting goal scheduling mechanism, as well as the priority goal scheduling. It also shows two kinds of goal distribution mechanisms: for tightly coupled multiprocessors with shared memory, and for loosely coupled multiprocessors.

Memory management: An essential role for logic programming is to free programmers from having to perform the memory management. In other words, the KL1 implementation has to include efficient memory management schemes. Section 4 shows the incremental garbage collection mechanism embedded in the parallel KL1 implementation.

Distributed unification: We have designed the principal operation (unification) in distributed environments. Section 5 discusses how to reduce the communication cost in distributed unification.

KL1-B: The above schemes for the KL1 parallel implementation are condensed into the abstract instruction set, called KL1-B (Kimura and Chikayama 1987).

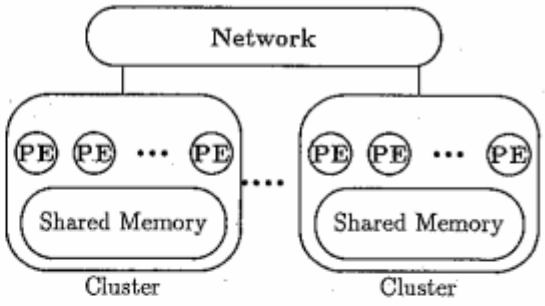

Figure 2: Abstract PIM Configuration

KL1-B interfaces PIMs and KL1, just as WAM (Warren 1983) interfaces Prolog and sequential machines. In other words, KL1-B represents the abstract architecture of PIM. Section 6 overviews the KL1-B features.

Hardware architecture of the PIM pilot machine: The hardware architecture of the PIM pilot machine is shown in section 7. We introduced a hierarchical configuration into the PIM hardware architecture (shown in Figure 2), which is assumed in the above discussions about KL1 parallel implementation. Each processing element (PE) has a tagged architecture. Several PEs form a *cluster*. All PEs in a cluster can share a memory space which is local to each cluster. These clusters are interconnected by a communication network. A shared memory in each cluster works as a local memory for inter-cluster parallel processing. In other words, intra- and inter-cluster addressing systems are separated.

The Multi-PSI (Taki 1986) 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 (Nakashima and Nakajima 1987) connected by the fast mesh network (Takeda et al. 1988). From the KL1 implementation viewpoints, each processing element in the Multi-PSI can be seen as a cluster of one processor. Most of the KL1 implementation issues in distributed environments have been studied through the design of the Multi-PSI system (Ichiyoshi et al. 1987).

2 RESOURCE MANAGEMENT BY SHŌEN

KL1 was initially specified as flat GHC (Ueda 1986a, Ueda 1986b), taking efficient implementation into consideration. Flat GHC is a subset of GHC, which allows only built-in predicates as guard goals. This restriction makes language implementation more efficient while retaining most of GHC's descriptive power. Starting from flat GHC, KL1 has been extended so that it has become a practical language with the features required for the PIMOS design[1].

[1]Chikayama et al. (1988) describe the system programming features in KL1. Also Miyazaki (1988) in Japanese.

2.1 Metaprogramming by Shōen

In GHC or flat GHC, 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 metaprogramming capability into KL1.

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

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

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

A shōen 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 shōen. All forked goals from the given *Goal* belong to the same shōen. *Min* and *Max* are minimum and maximum possible priorities of goal scheduling allowed in the shōen. (See section 3.3.)

Control and *Report* are the control and the report streams. The control stream is used to start, stop or abort the shōen from outside. The monitoring process can be informed of events within a shōen such as the end of execution and exceptions through the report stream. Exceptions that have occurred in the shōen or are delegated from one of the child shōens are reported as a message to the report stream. *Mask* is a bit pattern for determining which exceptions should be handled in this shōen. 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 shōen. Any kind of failure is treated as an exception. The logical conjunction between KL1 goals is maintained within each shōen. In other words, goals in a shōen do not form a conjunction with goals outside the shōen.

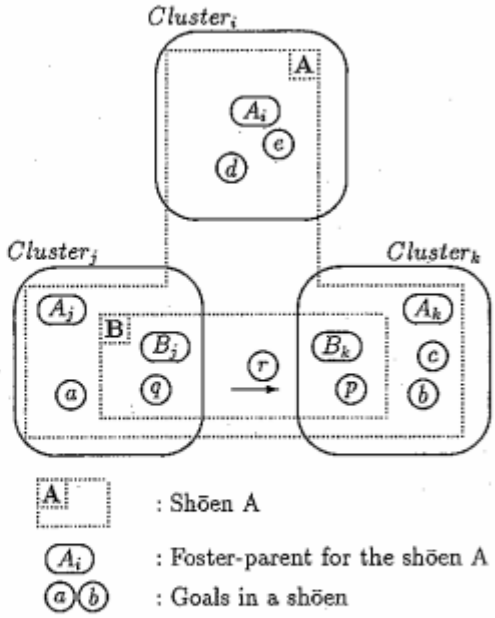

Figure 3: Shōen and Foster-parents

2.2 Distributed Resource Management by Foster-Parents

The main role of a shōen is to control the execution under the shōen, that is, the shōen 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 7.4) 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 could be very high and the synchronization would be very complicated. We provided a *shōen and foster-parent scheme* to avoid this (Ichiyoshi et al. 1987).

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 shōens or goals belonging to the shōen in that cluster, that is, 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 done by the communication between the children and the foster-parent within a cluster, so that the inter-cluster communication traffic can be reduced.

Figure 3 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). The 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.

2.3 Weighted Throw Count

Termination detection of all or some processes is one of the principal functions in any systems. The end of a KL1 program execution corresponds to the end of the shōen. When all goals in a shōen or descendant shōens 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 shōen 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 3.

The weighted throw count (WTC) method was provided to solve this problem (Rokusawa et al. 1988), where certain weight is assigned for the shōen, its foster-parents, and messages. The WTC can be seen as an application of the so-called weighted reference counting (Watson and Watson 1987, Bevan 1987).

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

$$W_{shoen} + \sum(W_{fosterparent}) + \sum(W_{message}) = 0$$

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 shōenwith 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.

3 GOAL SCHEDULING

3.1 Goal Reduction by Register Machines

While any unifications of KL1 can be done in parallel under the semantics of GHC (Ueda 1985), we did not adopt this fine-grained parallelism, but 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 into KL1-B code as shown in section 6, executed by single thread of control from guard to body. No parallelism is expected within each goal reduction.

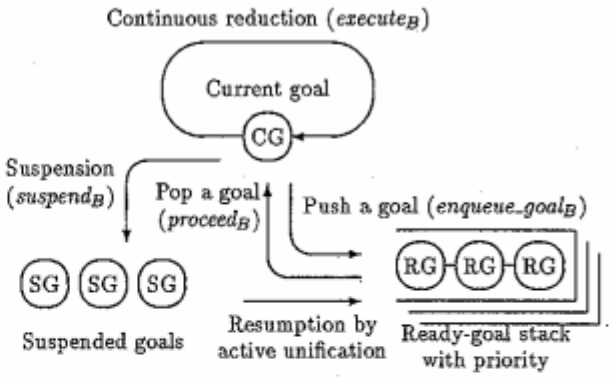

Figure 4: Goal State Transition and KL1-B Instructions

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 Non-Busy Waiting Goal Scheduling

A goal can be a *ready goal* (RG), a *suspended goal* (SG) or a *current goal* (CG), as shown in Figure 4. The *ready* goals are linked into a list forming a *ready-goal-stack*. In principle, a current goal is popped up from the ready-goal-stack, then the goal reduction is performed by 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 (Ichiyoshi et al. 1987, Sato et al. 1987). 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.

3.3 Priority Goal Scheduling and Pragmas

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 4), while other body goals are linked to the ready-goal-stack.

The priority of goal scheduling can be controlled by specifying pragmas (Shapiro 1984). While each shōen is created with the maximum and minimum priority (see section 2.1), the pragmas can specify the relative priority within the range allowed for the shōen. 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.

3.4 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. Although several ideas for load distribution

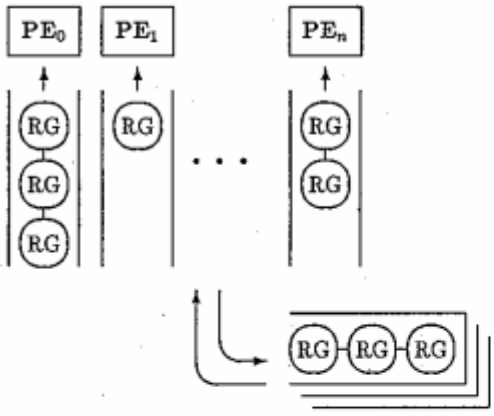

Figure 5: Goal Scheduling within a Cluster

have been proposed so far (Chikayama 1986, Takeda et al. 1988, Shapiro 1984), we should continue to study how to distribute the processing load. Currently the following strategies are provided in the KL1 implementation on PIM.

In a cluster, we provided an individual ready-goal-stack for the goals with highest priority on each processing element, as shown in Figure 5, to avoid conflicts of access to the common goal-stack (Sato et al. 1987). The highest-priority goals are distributed 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 (Sato and Goto 1988). In the on-demand scheme, an idle processor sends a request to a busy processor. On receiving the request, the busy processor sends the goal from its cache in the ready-goal-stack to the idle processor. These communication should be done efficiently within a cluster, so that we designed a coherent cache and an inter-processor signaling by *slit-checking* for the PIM pilot machines. (See section 7.2.)

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, saving the goals in each ready-goal-stack into the common ready-goal-stack.

3.5 Inter-cluster Goal Distribution

The load distribution among clusters should be done carefully because the communication cost is more expensive than within a cluster. Therefore, we provided pragmas by which users can give the indication for load distribution.

The pragmas for load distribution have 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. If there is no foster-parent, one will be created on the spot. In the future, we plan to implement a dynamic load-balancing mechanism.

4 MEMORY MANAGEMENT BY MRB

4.1 Importance of Efficient Garbage Collection

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 (Clark and Gregory 1984, Shapiro 1983, Ueda 1986b) 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 (Warren 1983), 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 implementations.

4.2 Incremental Garbage Collection by MRB

Reference counting (Cohen 1981) is one method by which to recognize incrementally when a certain storage area has become inaccessible from the program. However, in reference counting, each word cell must have a reference counter field for the whole memory space. In addition, the cost of updating the reference counter is high, because data objects must always be accessed.

Several methods were proposed to reduce these overheads relying on the fact that data objects are not used very many times, and most are used only once (Deutsch and Bobrow 1976). Multiple reference bit (MRB) method was proposed as an incremental GC method for concurrent logic programming languages[2] (Chikayama and Kimura 1987).

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

[2]Another incremental GC method called lazy reference counting (LRC) (Goto et al. 1988) was designed. LRC uses two-word indirect pointers with a reference counter.

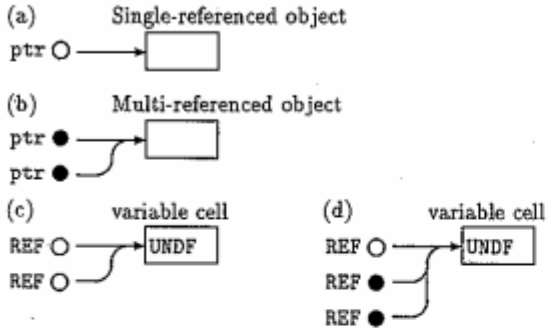

Figure 6: References in the MRB Scheme

in the pointers rather than in the pointed objects, no extra memory access is required for reference information maintenance.

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, *off-MRB* by ○ and *on-MRB* 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 off-MRB, 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 off-MRB, 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 *off-MRB*, the storage area for the cons cell can be reclaimed during the 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 run-time overheads. The MRB scheme also makes available several optimization techniques, such as destructive array element update without using the method in Barklund and Millroth (1987).

4.3 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 garbage collection that is done locally within a cluster accompanied with the incremental garbage collection by MRB. This is because the MRB scheme leaves some garbages. We first implemented a simple garbage collection of so-called *copying* scheme on our experimental KL1 system.

We designed the parallel mechanism to collect garbages by all processing elements in a cluster. When a certain processing element finds the shortage of memory space during its goal reduction, it informs this event to other processing elements, after it finishes the current goal reduction. This is because garbage collection is difficult to start during a goal reduction. So, the shortage of memory space should be detected before all memory area is used up. After all processing elements stop their goal reductions, they start the copying operations tracing all active cells in a shared memory of a cluster. Here, the copying roots are the ready goals in ready-goal-stacks[3].

We also studied garbage collection schemes tailored to the KL1 parallel processing, and designed a new scheme called *Piling* garbage collection (Nakajima 1988). The piling scheme has the feature of *life-time* (Lieberman and Hewitt 1983). The piling scheme can be used with the MRB scheme, as well as can be done in parallel by all processing elements in a cluster.

5 DISTRIBUTED UNIFICATION

5.1 Export/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 shōen to which the goal belongs. The encoding of arguments (or any KL1 data) is called *exportation*; decoding is called *importation*.

In the KL1 parallel implementation a reference can be *external* or *internal*. An external reference is a reference to a non-local data. The external reference 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 identification number of location of the data in that cluster.

We did not choose to take the memory location directly as the unique identification number, *ent*, because

[3]Export tables in section 5 are also the roots of copying operations.

that would make it very difficult to do garbage collections locally within one cluster. Ordinary garbage collections by marking or moving schemes are sometimes required even if the MRB incremental garbage collection is adopted. If the locations of data have moved as the result of these garbage collections, 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 (Ichiyoshi et al. 1987). 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 the 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.

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 the import table entry and vice versa. Export and import tables are shown in Figure 7, where an external reference cell is indicated by EX cell[4].

5.2 Avoiding Duplicated Exportation/ Importation

When there are multi-referenced data objects in a cluster, they may be exported more than once. In such cases, each exportation tends to use export table entries. In addition, if a cluster imports the same data structure more than once, the cluster has to allocate its memory for the same data structure.

As multiple references are managed by the MRB scheme in each cluster, each exportation can find the possibility of multiple exportation for each data objects. Single-referenced data objects may not be exported more than once. Therefore, we introduce two kinds of export and import tables, each for single-referenced objects and multi-referenced objects. Slightly complicated procedures are introduced for multi-referenced objects to save export table entries and to avoid duplicated importations, while a simpler external reference mechanism is used for single-referenced objects.

A hash table is attached to the export table for multi-referenced objects. In case a multi-referenced 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 exportations. There is also a hashing mechanism for retrieving an import table entry for multi-referenced objects 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.

[4]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.

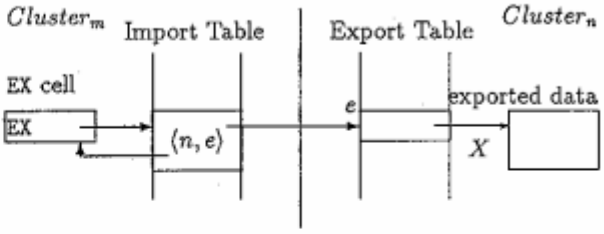

Figure 7: Export Table and Import Table

The introduction of export and import tables help reduce the number of inter-cluster read requests as follows. Suppose $Cluster_n$ 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 exportations (by export table mechanism with hashing), $Cluster_m$ allocates only one external reference cell to X (by 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 *%read* message to the referenced cluster.

%read(X, ReturnAddress)

Where X is the external reference $\langle n, e \rangle$ in Figure 7, and *ReturnAddress* is a newly created export table entry $\langle m, i \rangle$ for returning the value[5].

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

%answer_value(ReturnAddress, V)

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:

[5]The *%read* and *%answer_value* messages correspond to the *%read_value* and *%return_value* messages in Ichiyoshi et al. (1987).

%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 does 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 by: first compare the two cluster identifier, then make reference pointers always in the same direction, in descending order (or ascending order) of cluster identifier (Ichiyoshi et al. 1988).

5.4 Distributed Garbage Collection by WEC

Since export table entries for multi-referenced data objects cannot be freed by a local garbage collection within a cluster written in section 4.3, there must be an inter-cluster garbage collection mechanism to free those entries that have become garbage.

One way of realizing inter-cluster garbage collection is by a *global garbage collection*. We are designing a parallel mark-and-collect type global garbage collection. A serious problem with global garbage collection is that it will take a very long time.

Another is an incremental inter-cluster garbage collection. The merit of such a garbage collection scheme is that it keeps intact the locality of data access in the program. A naive implementation of the standard reference counting scheme, however, does not work correctly in a distributed environment.

Unlike the standard reference counting which assigns reference counts to only referenced data, the weighted export counting (WEC) scheme assigns reference counts, or weighted export counts (*wec*), to references (pointers) as well as to referenced data (Ichiyoshi et al. 1988). 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 invariant is kept true for every export table entry E (See Figure 8.):

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

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*.

The WEC technique has been used in functional language implementations on multiprocessors (Watson and Watson 1987, Bevan 1987), but we introduced it for the incremental garbage collection of export table entries.

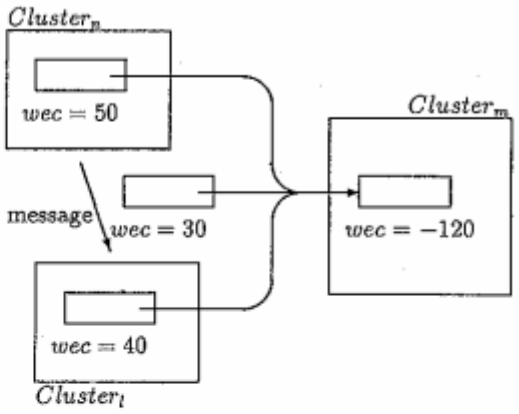

Figure 8: WEC Invariant

6 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[6]. It was extended for parallel execution.

The role of KL1-B is similar to that of WAM (Warren 1983). The major differences are:

- Passive unification instructions will be suspended when instantiation of variables is required to accomplish the unification.
- 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.

Most instructions in KL1-B include run-time data type checks. The actions that follow the run-time type check are very different.

6.1 Data Type and Goal Record

All the memory words and all the argument/temporary registers can 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. The *type* shows the data type information such as:

`UNDF`: Undefined variable.

[6] An explanation of each KL1-B instruction can be found in Chikayama and Kimura (1987), and Kimura and Chikayama (1987).

```
Label1:   Code for the first clause
          ( Head and guard part: )
              passive unification instructions
              (commit)
              garbage collection instructions
          ( Body part: )
              active unification instructions
              argument preparation instructions
              goal fork instructions
Label2:   Code for the second clause
          ...
          ...
Labeln:   Code for the last clause
          ...
Labeln+1: suspendB pred
```

Figure 9: A Form of Compiled Codes in KL1-B

`HOOK`: Undefined variable, some goals are waiting for instantiation of this variable. The value is the pointer to these goals.

`REF` : Indirect pointer or reference pointer to an undefined variable cell.

`INT` : Integer.

`ATOM`: Symbolic atom.

`LIST`: List cell.

`EX`: External reference pointer.

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 some control information. The argument list includes atomic values or pointers to variables or structure bodies in the heap.

6.2 Compiled Code in KL1-B

A set of candidate clauses for a predicate is compiled into a sequence of KL1-B instructions[7] as in Figure 9. A goal reduction is initiated by a KL1-B instruction[8], $proceed_B$, popping up 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). 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.

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

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

Table 1: Passive Unification Instructions

KL1-B Instruction	Comment
(For goal arguments)	
$wait\_value_B$ Ai, Aj	Unify two instantiated terms.
$wait\_const_B$ $Ai, Const$	Wait a constant $Const$ in Ai.
$wait\_list_B$ Ai	Wait a list in Ai.
$wait\_vect_B$ $Ai, Arity$	Wait an $Arity$ vector in Ai.
(For structure elements)	
$read\_car_B$ Ai, Aj	Read car of a list Ai into Aj.
$read\_cdr_B$ Ai, Aj	Read cdr of a list Ai into Aj.
$read\_element_B$ Ai, N, Aj	Read the N-th element of a vector Ai into Aj.
(Indexing)	
$switch\_on\_type_B$ $Ai, Label_{Const}, Label_{List}, \ldots$	
$branch\_on\_const_B$ $Ai, Entry\_Table$	
(Suspension and labels)	
$try\_me\_else_B$ $Label$	Set a branch label $Label$.
$suspend_B$ $Goal$	Suspend $Goal$.

Note: Ai and Aj are the argument registers.

```
wait_const_B Ai, Const:
    put the dereferenced result of Ai to Ai
    check the equality between Ai and Const
    if they are equal then proceed to the next code
    else if Ai is uninstantiated
                or an external reference
            then push Ai to the suspension stack
        jump to Label
```

Figure 10: A KL1-B Instruction: $wait\_const_B$

A KL1-B code for a set of candidate clauses includes passive unification instructions for head and guard part, active unification instructions, argument preparation instructions and goal fork instruction for body part, and garbage collection instructions.

6.3 Passive Unification

Table 1 shows typical passive unification instructions and a suspension instruction in KL1-B. They 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.

Figure 10 shows the action of a passive unification instruction, $wait\_const_B$. $Wait\_const_B$ corresponds to a passive unification between a current goal argument, Ai, and a constant value, $Const$. $Label$, indicated by the preceding $try\_me\_else_B$, is a branch address when the passive unification is suspended or failed.

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

Table 2: Garbage Collection Instructions

KL1-B Instruction	Comment
$mark_B$ *Ai*	Set MRB of *Ai* on.
$collect\_value_B$ *Ai*	Reclaim along the reference from *Ai*.
$collect\_list_B$ *Ai*	Reclaim a list cell *Ai*.
$collect\_vect_B$ *Ai*	Reclaim a vector *Ai*.

content is an indirect pointer or not. If it is an indirect pointer, the pointed cell is dereferenced until some 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.

The *%read* message is not sent, in principle, in the 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. This is because other candidate clauses may be committed.

6.4 Suspension

If no clause is selected for the current goal, $suspend_B$ instruction finally tests the suspension stack. If there is no variable, an exception of failure 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, *%read* message is sent to the node where the exported data resides (see Figure 7). 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 by *%answer_value* message. However, the exported data may be an unbound variable cell. In this case, the action of replying 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 *%answer_value* message.

6.5 MRB Maintenance and Garbage Collection

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

The MRB is maintained in each KL1-B instruction.

collect_list$_B$ *Ai*:
  if MRB of *Ai* is *off*
    then reclaim the cons cell pointed by *Ai*
    else proceed to the next instruction.

Figure 11: A KL1-B Instruction: $collect\_list_B$

```
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 12: An Active Unification Example

In addition, several garbage collection instructions are introduced to KL1-B. (See Table 2.) The compiler detects candidate places where reference paths are added. In this case, $mark_B$ is used to set MRB *on*[9]. 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. $Collect\_list_B$ in Figure 11 is a typical KL1-B instruction which corresponds to the goal reduction by, for example, the clause:

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

This clause unifies the goal argument with a cons cell, then retrieves its elements X and Y. In this case, one reference path to the cons cell is consumed, if the clause is committed. Therefore, the garbage collection instruction, $collect\_list_B$, is executed after the head and guard part execution ends successfully. The $collect\_list_B$ reclaims the cons cell if it is a single-referenced cell (*off-MRB*).

6.6 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*. The KL1-B instructions in Table 3 and 4 are provided for them. Figure 12 shows the typical compiled code for the active unification in such a clause as:

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

The structures for the 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. (See Figure 12.) 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

[9] $Mark_B$ is merged with the argument preparation instructions in Table 3.

Table 3: Active Unification Instructions

KL1-B Instruction	Comment
⟨ Active unification ⟩	
get_value_B Ai, Aj	Unify Ai and Aj.
get_const_B $Ai, Const$	Unify Ai and $Const$.
$get\_list\_value_B$ Ai, Aj	Unify Ai and a list (Aj).
$get\_vect\_value_B$ Ai, Aj	Unify Ai and a vector (Aj).
⟨ Argument preparation ⟩	
$put\_variable_B$ Ai, Aj	Make a new variable pointed by Ai and Aj.
put_value_B Ai, Aj	Move a variable from Ai to Aj.
put_const_B $Ai, Const$	Write $Const$ in Ai.
put_list_B Ai	Allocate a list cell in Ai.
put_vect_B $Ai, Arity$	Allocate an $Arity$ vector in Ai.
⟨ Argument preparation in a forked goal ⟩	
$set\_variable_B$ Gi, Aj	Gi is the i-th argument of a forked goal.
set_value_B Gi, Aj	
set_const_B $Gi, Const$	
set_list_B Gi	
set_vect_B $Gi, Arity$	
⟨ For structure elements ⟩	
$write\_car/cdr\_variable_B$ Ai, Aj	
$write\_element\_variable_B$ Ai, N, Aj	N: the element position in a vector.
$write\_car/cdr\_value_B$ Ai, Aj	
$write\_element\_value_B$ Ai, N, Aj	
$write\_car/cdr\_const_B$ $Ai, Const$	
$write\_element\_const_B$ $Ai, N, Const$	

```
get_list_value_B Ai, Aj:
    put the dereferenced result of Ai to Ai
    if Ai is uninstantiated
      then if Ai is linked by suspended goals
             then resume suspended goals
           Ai := Aj and proceed to the next code
      else if Ai is an external reference
             then send %unify message
             else if Ai is list
                    then do general unification
                           between Aj and Ai
                    else Failure
```

Figure 13: A KL1-B Instruction: $get\_list\_value_B$

Table 4: Goal Fork Instructions

$proceed_B$	
$execute_B$	$Goal$
$enqueue\_goal_B$	$Goal$
$enqueue\_with\_priority_B$	$Goal, Priority$
$enqueue\_to\_processor_B$	$Goal, Node$

reclaim a structure body and its elements at 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 12, $get\_list\_value_B$, is a typical KL1-B instruction for active unification. This instruction has one of four kinds of actions, selected by checking the data type, as in Figure 13.

When Ai is an uninstantiated variable without suspended goals, that is, the tag of Ai is UNDF, Aj (a pointer to a cons cell made by the first instruction in Figure 12) is assigned into 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 (Sato et al. 1987). Here, it is important to shorten the period fo locking the unbound variable. Therefore, the compiler generates the compiled code as in Figure 12, where the right-hand-side structure is created first. As a result, the unbound variable is locked only within $get\_list\_value_B$ instruction.

If Ai is an uninstantiated variable with suspended goals, that is, the tag is HOOK, 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 4.) 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: the tag is EX. 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, the unification fails and an exception occurs.

6.7 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. (See Table 4.) As in Figure 4, 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 continue just after the current goal reduction, while other goal(s), r, is pushed into the ready-goal-stack. The KL1-B code for the above clause will be as follows.

Head and guard execution for p.
...(commit) ...
Arguments preparation for a goal record r.
$enqueue\_goal_B$ r
Arguments preparation in registers for q.
$execute_B$ q

The KL1-B instruction $execute_B$ q is a jump operation to the top of KL1-B code for the goal q.

Other body goals are pushed by $enqueue\_goal_B$ instructions. When scheduling priority was specified by the pragmas, the KL1 compiler generates a KL1-B instruction, $enqueue\_with\_priority_B$. When the pragmas for load distribution were 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 incidentally happen in KL1 execution: a garbage collection requirement (section 4.3), an inter-processor communication request, and a goal fork with the highest priority (section 3.4. These events are only detected by *slit-checking* in $execute_B$, $proceed_B$ and $suspend_B$ instructions, that is, 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 highest priority, the corresponding actions do not have to be performed immediately. So, they may be delayed until after the goal reduction finishes.

As in section 2.2, a foster-parent in a cluster holds the shōenstatus as well as the information about the computing resources assigned for the foster-parent. Before $execute_B$, $proceed_B$ or $suspend_B$ start a goal reduction, they check the shōenstatus of a current goal, and the computing resources left in that foster-parent.

7 HARDWARE ARCHITECTURE OF PIM

7.1 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 one hundred times higher performance than conventional machines. To achieve this goal, we studied new processing element architectures as well as new parallel architectures to connect more than one hundred 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 include 128 processing elements. In the following, we would like to focus on the hardware architecture of the PIM/p.

7.2 The Pilot Machine: PIM/p

7.2.1 Hierarchical Structure in PIM/p

In the parallel architecture design for the PIM/p, we aimed 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 14. Eight processing elements (PEs) form a cluster with shared memory. The PIM/p consists of 16 clusters connected by inter-cluster network.

7.2.2 Processing Element Design Issues

The PIM/p processing element is newly designed for the efficient implementation of KL1. The design started by analyzing the behavior of the KL1-B instructions (Shinogi et al. 1988).

As discussed in section 6, run-time data type checks are essential for KL1-B instructions. So we introduced the tagged-architecture to the CPU design. The next issue is how to implement the polymorphic functions in KL1-B[11], 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 has advantages in hardware design cost. 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 (Nakashima and Nakajima 1987). Therefore, we designed the RISC-like instructions with the conditional macro-call instructions for the PIM/p processing elements, so that both the advantages in the RISC-like instructions and microprogrammable processors are available in the KL1-B implementation on the PIM/p.

The principal operations such as the incremental garbage collection by MRB and dereferencing are supported by the dedicated RISC-like instructions. Section 7.6 discusses the instruction set and corresponding KL1 features. To shorten the machine cycles, the CPU was designed to execute the RISC-like instructions by four-stage pipeline. The processing element performance estimated from the compiled code is over 600 K

[10] RPS: KL1 goal reductions per second
[11] The detailed discussion can be found in Shinogi et al. (1988).

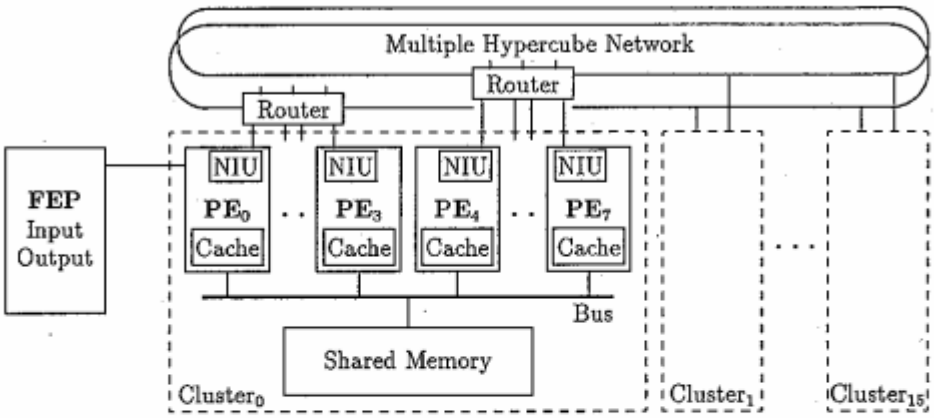

Figure 14: The Pilot Machine: PIM/p

RPS for the append program. Note that the estimated performance includes the incremental garbage collection cost using MRB. The detail configuration of the PIM/p processing elements is shown in section 7.3.

In addition to the KL1-oriented instructions, the PIM/p processing element includes the functions for the KL1 parallel implementation: an interrupt mechanism for the slit-checking, a coherent local cache tailored to the characteristics of KL1, and the network interface for inter-cluster parallel processing.

7.2.3 Design Issues for Cluster

Processing elements within each cluster share one address space. Focusing on KL1 parallel execution in each cluster, quick and exclusive access to shared data is a key issue. We designed a local coherent cache protocol for the KL1 parallel processing. Each processing element in the PIM/p has a coherent cache memory designed specifically for KL1 parallel execution (Matsumoto et al. 1987). The cache mechanism increases not only the efficiency of local execution on each processing element, but enables high-speed communication within a cluster. It is also necessary to provide an efficient mechanism to access shared data exclusively. The exclusive memory access can be obtained at a low cost by using the cache block status of the coherent cache memory. (See section 7.4.)

7.2.4 Design Issues for Inter-Cluster Network

As discussed in section 5 and 6, inter-cluster communication will possibly be required during a unification instruction of KL1-B on each processing element. That communication may include various kinds of messages. We aimed at the followings in the design the inter-cluster network:

- Enough performance for both short and long message packets.
- Inter-cluster processing where it is required.

The hyper-cube structure is 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. Each processing element has a network communication port to send and receive messages between clusters, so that inter-cluster communication operations can be done on the spot. The network router and the network interface unit on each processing element are written in section 7.5.

7.3 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 as shown in Figure 15. The target of the basic machine cycle is 50 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 memory redundancy can be useful to detect a cache memory error, because ECC is not adopted in the cache. The cache controller units (CCU) manage both the instruction cache and the data cache. The cache address array would be updated by both commands from the CPU and a common bus. To avoid the access conflict, the CCUs include two cache address arrays with cache block status.

The CPU has two instruction streams, one is from the instruction cache, and the other is from the internal instruction memory (IIM). The IIM is similar to a writable microprogram store. Hopefully, the CPU will execute an instruction at every 50 nanoseconds using a four-stage pipeline in most cases. The CPU has two co-processors: a network interface unit (NIU) and a floating point processor unit (FPU). The CPU has a common protocol to use both co-processors.

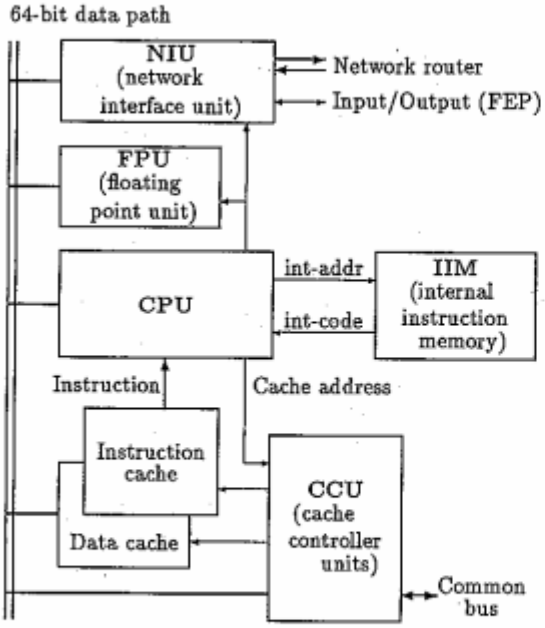

Figure 15: PIM/p Processing Element Configuration

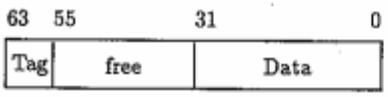

Figure 16: KL1 Tagged Data Representation

7.3.1 Memory Model

The PIM/p has a 4G-byte global virtual address space on each cluster. Taking practical KL1 implementation into consideration, 40-bit KL1 data (an 8-bit tag and 32-bit data) is necessary. However, we decided not to build a complete 40-bit system, because: (1) it may be difficult to use an off-the-shelf memory system as shared memory; and (2) instructions should not necessarily be placed on the 40-bit boundary. Although KL1 data density will be low in the PIM/p memory system[12], this will not cause performance degradation. Normal KL1 data is placed by 40-bit KL1 tagged data in aligned 64-bit words in the PIM/p memory system, as shown in Figure 16. Instructions and some data structures, such as strings or floating point numbers, are placed on a byte boundary.

7.3.2 CPU Execution Pipeline

A PIM/p processing element has two kinds of instructions, external and internal, but most of them are common. *External instructions* are used to represent compiled codes of user programs. They include KL1 support instructions as well as simple RISC-like instructions. *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. They are invoked by external macro-call instructions.

[12] As an alternative to MRB garbage collection, LRC(lazy reference counting) (Goto et al. 1988) is now being examined. In LRC method, the free three bytes in each tagged data will be used as a reference count field.

Table 5: Pipeline Stage and its Operation

	Operation
D	Decode / Address register read
A	Address calculation
T	Data register read / Cache address access
B	ALU / Cache data access / Register write

The processing element uses an instruction buffer and a four-stage pipeline, **D A T B**, to attempt to issue and complete an external instruction every cycle. External instructions are either four, six or eight bytes long, so that the instruction buffer has a hardware aligner. Each internal instruction requires two additional stages, preceding stage **D**, to set the internal instruction address (stage **S**) and to fetch the instruction (stage **C**).

Table 5 shows the pipeline stages in both ALU and memory access instructions. General-purpose registers are updated only at the last **B** stage, thereby avoiding write conflicts. Internal forwarding is done by hardware so that the result of a register-to-register instruction can be used by the next instruction even though that result has not yet been written to the general registers.

In a branch instruction to an external instruction, the branch target instruction is fetched at stage **B** in the same way as memory read instructions. Therefore, ordinary branch instructions may cost three additional cycles to branch. Delayed branch instructions can avoid the three cycles by executing other effective instructions.

Most tag branch instructions test their condition at stage **B**. However, macro-call instructions and some internal branch instructions test their condition at stage **A**. Figure 17 shows the macro-call instruction pipeline. A macro-call instruction initiates the internal instruction fetch (stage **S**) at its stage **D**, then tests its condition at stage **A**. Therefore, even if the branch is taken, a macro-call instruction costs only one additional cycle to invoke a subroutine in the IIM. In addition, delayed macro-call instructions are provided to avoid the penalty. Return from macro-call, that is, return from internal instructions to external instructions, can be indicated by a one-bit flag: *eoi*. The internal instruction memory has an *eoi* field for each instruction, so that the execution of the macro body will finish at any internal instruction except for branch instructions. (See Figure 17.)

```
When the condition is true:
D  A  (condition test at A)   : macro-call instruction
   D  (canceled)              : next external instruction
S  C  D  A  T  B              : first internal instruction
   S  C  D  A  T  B           : second internal instruction

When the condition is false:
D  A  (condition test at A)   : macro-call instruction
   D  A  T  B                 : next external instruction
      D  A  T  B              : external instruction

End of macro body:
S  C  D  A  T  B              : internal instruction eoi
   S  C  (canceled)           : internal instruction
      S  (canceled)           : internal instruction
         D  A  T  B           : next external instruction
```

Figure 17: Macro-call Instruction Pipeline

7.3.3 Registers

The processing element includes 32 general-purpose registers with some dedicated registers. These registers are specified by a 6-bit register specifier in most instructions. 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, a slit-check register (see sections 6.7 and 7.3.4), and a tag mask register to mask tag fields in conditional branch instructions. 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. (See section 7.6.5)

Internal instructions can use virtual registers, called indirect registers, in addition to the above registers. Through the indirect registers, internal instructions can 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.

In addition to the above registers, the processing element has co-processor registers, which are handled only by co-processor interface instructions.

7.3.4 Slit-check and Interrupt

A hardware mechanism for *slit-checking* (see section 6.7) 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, the slit-checking might be implemented using normal interrupt mask/unmask operations and cumbersome interrupt handler. It would cost too much for the KL1 system. By incorporating the hardware slit-checkinf mechanism, the processing element can avoid frequent mask/unmask operations and interrupt handling overhead.

Table 6: 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 cache misses at block boundary, write data into 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.

7.4 Cache System

The design of a local coherent cache is a key issue to increase the efficiency of local execution on each processing element, and it enables high-speed communication within a cluster. Several coherent cache protocols have been proposed so far (Archibald and Baer 1986, Goodman 1983, Bitar and Despain 1986, Papamarcos and Patel 1984). Here, reducing common bus traffic is a more important design issue than reducing cache miss ratio (Goodman 1983).

We aimed to design a cache protocol for KL1 parallel execution. We developed a coherent cache simulator with a KL1 experimental system. The local coherent cache for PIM/p is designed based on the simulation result (Matsumoto et al. 1987). 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 cache 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 6 shows the basic CPU commands to the cache.

7.4.1 Cache Commands for KL1 Support

In parallel implementation of KL1, some data structures can be known when they are not accessible. A typical ex-

ample 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. So, when the receiver processor makes a message in the same area, cache invalidation of another cache will occur. The CPU command, *Read_Invalidate*, is provided to avoid such invalidation by invalidating at cache-to-cache data transfer.

In normal write operations, *fetch-on-write* is used. However, when new 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 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 the own cache block just after CPU reads the last cache block word, so that *Direct_Write* command can be used for already-used memory area.

7.4.2 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 6.6) or to link suspended goals to them (see section 6.3). Although lock conflicts seldom occur, lock latency is high in KL1 execution. The simulation results in Matsumoto et al. (1987) shows that th *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 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.

7.4.3 Cache Configuration

The capacity of both the instruction and data caches is 64K bytes. In general, a larger cache is necessary to maintain a high hit-ratio. However, it is preferable to give up forming a large cache by enlarging the cache block size. This is because our software simulation results have found that a cache block larger than four tagged words causes an increase in shared blocks between caches in parallel execution of KL1, so that mutual cache invalidation may increase (Matsumoto et al. 1987). On the other hand, the size of the cache address array is restricted by the LSI capacity of the cache controller unit (CCU). Therefore, the CCU has a block status tag for each 32-byte (four tagged words) block, and an address tag for each two blocks, that is, every 64 bytes. Our simulation result also shows that that scheme does not decrease the performance so much compared to a full 32-byte block cache of the same capacity.

7.5 Hyper-Cube Network and Network Interface Unit

The hyper-cube structure (Broomell and Heath 1983) is 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 distributedly implemented on each cluster.

The network was designed aiming at the inter-cluster communication throughput of 40 M bytes/second. We chose the following configuration considering the limitations in hardware implementations. A network router was designed for six-dimension hyper-cube connection. While four dimensions are enough to connect 128 processing elements (16 clusters), the router switch will be available for the future extension. Each communication path has the throughput of a 20 M bytes/second, one byte every 50 nanoseconds, in both directions. To enable the 40 M bytes/second throughput, the inter-cluster network is 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 informed to the CPU by slit-checking mechanism. Therefore, these message handling operations can be done on each processing element.

7.6 PIM/p Instructions and Corresponding KL1 Features

This section focuses on the PIM/p instruction set and several important points in its design. Table 7 lists the notation for instruction operands.

7.6.1 Basic Memory Access and Tag Handling

Table 8 shows the basic memory access instructions. Each memory access instruction reads data to a destination register from a memory location whose address is specified by a register and immediate offset, and vice versa. The transferred data width can be 8, 16, 32 bits, 32 bits with an 8-bit tag, or 64 bits. 64-bit data is loaded to (or stored from) two neighboring registers. The memory access instructions for 64-bit data are useful to load

Table 8: Memory Access Instructions

Instruction	*Operands*	*Comment*
Read	Rd, Ra, ofst	Read tag and 4-byte data
ReadB/HW/W/DW	Rd, Ra, ofst	Read 1, 2, 4, 8-byte data
Write	Rs, Ra, ofst	Write tag and 4-byte data
WriteB/HW/W/DW	Rs, Ra, ofst	Write 1, 2, 4, 8-byte data
WritewTag	Rs, Ra, ofst, imtg	Write 4-byte data giving a new tag
PUSH	Rs, Ra, ofst	Push data into a free list
PUSHwTag	Rs, Ra, ofst, imtg	Push data giving a new tag
POP	Rd, Ra, ofst	Pop up data from a free list
POPwTag	Rd, Ra, ofst, imtg	Pop up data giving a new tag
MRBorRead	Rd, Ra, ofst	Read data with mrb OR
DEREF	Rd, Ra, ofst	Pop up data with mrb OR
DirectWrite/B/HW/W/DW	Rs, Ra, ofst	Write data in Direct_Write cache mode
ReadPurge	Rd, Ra, ofst	Read data, followed by cache purge
ReadInvalidate	Rd, Ra, ofst	Read data, invalidating other cache
ExclusiveRead	Rd, Ra, ofst	For the last cache block word: ReadPurge For other words: ReadInvalidate
LockRead	Rd, Ra, ofst	Lock address and read data
WriteUnlock	Rs, Ra, ofst	Write data and unlock address
Unlock	Ra, ofst	Unlock address

Table 7: Notation for Instruction Operands

Six-bit register specifier	
Rs,Rs1,Rs2	Source registers
Rd	Destination register
Ra	Base address register
Rt1,Rt2	Register for testing tag
R,R1,R2,..R5	Argument for macro-call
Immediate value	
imm(8/32/40)	Immediate constant
imtg(8)	Eight-bit immediate tag
ofst(8/16/24/32)	Immediate address offset
retofst(8)	Offset for return address
iaddr(16)	Internal memory address

and store the execution environment to and from a goal-record in $proceed_B$ or $suspend_B$.

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, as follows.

```
WritewTag Rs, Ra, offset, immTag;
          M[Ra+offset] ← data(Rs),
          M[Ra+offset+7] ← immTag
```

The memory access giving a new tag is a primitive operation in argument preparation instructions of KL1-B. Instructions to move the tag part of a register to the data part of another register, and vice versa, are provided as *register move* instructions in Table 9.

7.6.2 Support for Dereference and MRB Garbage Collection

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 listed in Table 8 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.

Their actions are specified as follows.

```
PUSH  Rs, Ra, offset: M[Ra+offset]←Rs,
                      Rs←Ra;
POP   Rd, Ra, offset: Rd←Ra,
                      Ra←M[Ra+offset];
```

Here, imagine ft to be the free list top pointer register:

```
POP r1, ft, -
```

allocates a cell to r1 from the free list pointed by ft, and:

```
PUSH ft, r1, -
```

links a cell to the free list pointed by ft from r1. The following POPwTag instruction is used to give a new tag.

```
POPwTag Rd, Ra, ofst, imtg
```

The POPwTag instruction is used to put a new tag in the register that has a pointer to a structure just allocated from a free list. For example, the KL1-B instruction, put_list_B, can be expressed as:

put_list$_B$: POPwTag r1, ft, -, LIST

The MRB of each pointer and data object has to 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 the indirect word cell has 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. The DEREF instruction acts as follows. Here, r1 is an argument register, and ptr is used to refer to an indirect word cell.

```
DEREF r1, ptr :
            ptr ← r1, r1 ← M[r1],
            mrb(r1) ← mrb(ptr) or mrb(r1).
```

7.6.3 Memory Access with Coherent Cache Control

As stated in section 7.4, the coherent cache of the processing element has the extended functions for KL1 parallel execution. The instruction set includes memory access instructions corresponding to each cache function: DirectWrite, ReadPurge ,ReadInvalidate, and ExclusiveRead, as shown in Table 8. 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.

7.6.4 ALU Instructions

Table 9 shows the instructions for data and tag computations. All ALU instructions have two source registers and one destination register. These instructions can be classified into three kinds: 32-bit data computation, 8-bit tag computation, and 40-bit computation. Although logical operations are available for both the tag and data, arithmetic operations and shift operations are limited to the data part.

7.6.5 Tag Branch Instructions

Table 10 shows branch instructions. Each external branch address is specified by the instruction pointer with address offset (ofst). The internal branch address is specified by the absolute address of the internal instruction memory.

The run-time test of the type tag is a primitive operation to implement KL1. As discussed in section 6, most unification includes a multi-way branch for the goal argument type. Some Prolog machines, such as the PSI (Nakashima and Nakajima 1987), 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, a tag-mask register is used to mask logical operation (see XorMask, NotXorMask in Table 10). To avoid frequent update of the tag mask register, some branch instructions have an immediate tag mask in their operands, such as Jump*Cond*ImmMask.

In the processing element of PIM/p, various hardware flags, such as the condition code of ALU operation and an interrupt flag, can be accessed as the tag of dedicated registers. Therefore, most conditional branch operations are performed as tag branch operations.

wait_list$_B$ *Ai, (Label)*:

```
if tag(Ai) is LIST  then proceed to the next code
  elseif  tag(Ai) is REF
      then put the dereference result of Ai to Ai
           if tag(Ai) is LIST
             then proceed to the next code
             elseif  Ai is uninstantiated
                  then push Ai to the suspension stack
                       and jump to Label
                  else  jump to Label
      else  jump to Label
```

Figure 18: A KL1-B Instruction: *wait_list*$_B$

7.6.6 High-Level Instructions Using Macro-Call

Macro-call instructions in Table 10 invoke small programs in the internal instruction memory (IIM) depending on given conditions. They are 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 microprogram. For example, the KL1-B instruction *wait_list*$_B$ in Figure 18 first tests the data type of a given argument. If the data type is the expected LIST, this instruction finishes. Otherwise, it selects the operation in Figure 18 according to the data type. A macro-call instruction corresponding to *wait_list*$_B$ is written as follows, where LIST is an immediate tag value and acp is an alternative

Table 9: ALU Instructions

Instruction	*Operands*	*Comment*
Dop	Rs1, Rs2/imm, Rd	Normal ALU operation
*Dop*40	Rs1, Rs2/imm, Rd	40-bit ALU operation
Shift	Rs, R/imm, Rd	Shift operation
AddwTag/SubwTag	Rs1, Rs2/imm, Rd, imtg	ALU operation giving a new tag
AddImm/LoadImm	Rsd, imm(32)	Add or load long immediate constant
SextB/HW	Rs, Rd	Sign extension
Top	Rs1, Rs2/imm, Rd	Tag computation
PEC	Rs, Rd	Priority encode
Move	Rs, Rd	Tag and data transfer
MoveTD	Rs, Rd	Move tag to data transfer
MoveDT	Rs, Rd	Move data to tag transfer

Note: *Dop*: Add, AddCarry, Subtract, SubtractCarry, AND, Or, Xor, NOT
*Dop*40: AND40, Or40, Xor40, XorMask40
Shift: ShiftLeft, ShiftRight, ShiftLeftDouble, ShiftRightDouble
Top : TagAnd, TagOr, TagXor, TagXorMask

Table 10: Branch Instructions

Instruction	*Operands*	*Comment*
External branch		
(Delay) Jump*Cond*	Rt1, Rt2/imtg, ofst	(Delay) Tag jump
(Delay) Jump*Cond*ImmMask	Rt1, Rt2/imtg, imtg, ofst	(Delay) Tag jump under immediate mask
(Delay) Jump*Cond*40	Rt1, Rt2, ofst	(Delay) 40-bit compare jump
sKip*Cond*	Rt1, Rt2/imtg, imm	Conditional skip
(Delay) Jump	Ra, ofst(32)	(Delay) Jump
(Delay) JAL	Ra, ofst(24), retofst	(Delay) Jump and link
Internal branch		
(Delay) MJump*Cond*	Rt1, Rt2/imtg, iaddr	(Delay) Tag jump
(Delay) MJump*Cond*A	Rt1, Rt2/imtg, iaddr	(Delay) Tag jump at A-stage
(Delay) MJump*Cond*ImmMask	Rt1, Rt2/imtg, imtg, iaddr	(Delay) Tag jump under immediate mask
(Delay) MJump*Cond*40	Rt1, Rt2, iaddr	(Delay) 40-bit compare jump
(Delay) MJump*Cond*40A	R1, R2, iaddr	(Delay) 40-bit compare jump at A-stage
MsKip*Cond*	Rt1, Rt2/imtg	Conditional skip
(Delay) MJAL	R, iaddr	(Delay) Jump and link
(Delay) MJump	iaddr	(Delay) Jump
Conditional macro call		
(Delay) MacroCall*Cond*	Rt1,Rt2/imtg,[R3,R4,R5] iaddr	(Delay) Macro call
(Delay) MacroCall	R1,[R2,R3,R4,R5,] iaddr	(Delay) Unconditional macro call

Note: *Cond* : And, NotAnd, Or, NotOr, Xor, NotXor, XorMask, NotXorMask

```
wait_type:  JumpNotXor @r0, REF, @r2;
            DEREF ptr, @r0;
            MJumpNotAnd @r0, UNB, case_unbound;
            MJumpNotAnd @r0, MRP, case_mrp;
            PUSH fr1, ptr;
            MJumpNotXorMask @r0, @d1, wait_type;
            Nop (eoi);
            ..........
```

Figure 19: *Wait_list*$_B$ and Internal Instructions

clause pointer register for *Label*.

```
MacroCallNotXorMask Ai, LIST, acp, wait_type;
```

The data type tag of register `Ai` is tested first. If the register `Ai` has a value with the `LIST` type, this macro-call instruction simply finishes. Otherwise, this macro-call instruction invokes an internal routine whose entry address is specified as *wait_type*. Figure 19 shows the internal instructions corresponding to *wait_list*$_B$. Here, `@r0` and `@r2` are indirect registers corresponding to arguments `Ai` and acp in the macro-call instruction. `@d1` is also an indirect register to show the immediate value in the second argument of the macro-call, namely, an immediate tag `LIST`. The first internal instruction, `JumpNotXor`, tests the tag of `@r0`, namely `Ai`. When the tag is `REF`, it proceeds to the next instruction for dereference. Otherwise, it jumps out to the external instruction specified by `@r2`, namely acp.

8 SUMMARY

We have described an overview of the parallel inference machine architecture. The 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 issues are implemented on the parallel software workbench, the Multi-PSI systems. We showed 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 research members in the FGCS project. Most ideas for the KL1 parallel implementations were born from the accumulation of their discussions.

We wish to thank all research members of the participant companies 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 the cooperative work with Mr. A. Hattori, Mr. T. Shinogi, Mr. K. Kumon and all of their colleagues at Fujitsu Limited. We also wish to thank the general manager of the Information Processing Division in Fujitsu Laboratories, Mr. J. Tanahashi, and the manager of the Artificial Intelligence Laboratory in Fujitsu Laboratories, Mr. H. Hayashi, for their useful comments.

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

REFERENCES

(Archibald and Baer 1986) J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation using a multiprocessor simulation model. *ACM Transaction of Computer Systems*, 4(4):273–298, 1986.

(Barklund and Millroth 1987) J. Barklund and H. Millroth. Hash Tables in Logic Programming. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 411–427, 1987.

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

(Bitar and Despain 1986) 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.

(Broomell and Heath 1983) G. Broomell and J.R. Heath. Classification Categories and Historical Development of Circuit switching topologies. *ACM Computing Surveys*, 15(2):95–133, 1983.

(Chikayama 1986) 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.

(Chikayama and Kimura 1987) 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.

(Chikayama et al. 1988) 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.

(Clark and Gregory 1984) 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.

(Cohen 1981) J. Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13(3):341–367, Sept. 1981.

(Deutsch and Bobrow 1976) L.P. Deutsch and D.G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *CACM*, 19(9):522–526, Sept. 1976.

(Goodman 1983) 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.

(Goto and Uchida 1985) 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).

(Goto and Uchida 1986) 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.

(Goto 1987) A. Goto. Parallel Inference Machine Research in FGCS Project. In *US-Japan AI Symposium 87*, pages 21–36, Nov. 1987.

(Goto et al. 1988) 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*, Seattle, WA, August 1988.

(Hwang and Briggs 1984) K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

(Ichiyoshi et al. 1987) 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.

(Ichiyoshi et al. 1988) N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura. 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.

(Kimura and Chikayama 1987) 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.

(Lieberman and Hewitt 1983) H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Base on the Lifetimes of Objects. *CACM*, 26(6):419–429, June 1983.

(Matsumoto et al. 1987) A. Matsumoto et al. Locally Parallel Cache Designed Based on KL1 Memory Access Characterstics. TR 327, ICOT, 1987.

(Miyazaki 1988) T. Miyazaki. Parallel Logic Programming Language KL1 – Its Implementation and an Operating System in It –. Transactions of the Institute of Electronics Information and Communication Engineers, J71-D(8), pages 1423–1432, August 1988 (In Japanese).

(Murakami et al. 1985) K. Murakami, K. Kakuta, R. Onai, and N. Ito. Research on Parallel Machine Architecture for Fifth-Generation Computer Systems. *IEEE Computer*, 18(6), June 1985.

(Nakajima 1988) K. Nakajima. Piling GC – Efficient Garbage Collection for AI Languages –. In *Proceedings of IFIP Working Conference on Parallel Processing*, Pisa, Italy, April 1988.

(Nakashima and Nakajima 1987) 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.

(Papamarcos and Patel 1984) 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.

(Rokusawa et al. 1988) K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. 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.

(Sato et al. 1987) 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.

(Sato and Goto 1988) 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.

(Shapiro 1983) E.Y. Shapiro. A subset of Concurrent Prolog and Its Interpreter. TR 003, ICOT, 1983.

(Shapiro 1984) 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.

(Shinogi et al. 1988) T. Shinogi, K. Kumon, A. Hattori, A. Goto, Y. Kimura, and T. Chikayama. Macro-call Instruction for the Efficient KL1 Implementation on PIM. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.

(Takeda et al. 1988) Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama, and K. Taki. 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.

(Taki 1986) 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.

(Ueda 1985) K. Ueda. Guarded Horn Clauses. TR 103, ICOT, 1985.

(Ueda 1986a) K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. TR 208, ICOT, 1986. (Also in Programming of Future Generation Computers, North-Holland, Amsterdam, 1987.).

(Ueda 1986b) K. Ueda. Introduction to Guarded Horn Clauses. TR 209, ICOT, 1986.

(Veen 1986) A.H. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

(Warren 1983) D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.

(Watson and Watson 1987) 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.