PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Architecture and Implementation of PIM/p

Kouichi KUMON Akira ASATO Susumu ARAI
Tsuyoshi SHINOGI Akira HATTORI
Fujitsu Limited
1015, Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

Hiroyoshi HATAZAWA
Fujitsu Social Science Laboratory Ltd.

Kiyoshi HIRANO
Institute for New Generation Computer Technology

Abstract

In the FGCS project, we have developed a parallel inference machine, PIM/p, as a one of the final outputs of the project [Taki 1992]. PIM/p has up to 512 processing elements(PEs) using two level hardware structures. Each PE has a local memory and a cache system to reduce bus traffic. The special cache control instructions and the macro-call mechanism reduce the common bus traffic, which may become the performance bottle-neck for shared-memory multi-processor systems. Eight PEs and a main memory are connected by common bus using the parallel cache protocol, we call it a *cluster*. PIM/p system consists sixty-four clusters, those are connected by dual sixth-order hyper-cube networks.

The KL1 processing system on PIM/p has two component, the compiler and the run-time support routines. The compiler uses the templates to generate PIM/p native codes from KL1-B codes. Each KL1-B instruction has a corresponding template. The codes are optimized after the expansion from KL1-B to native codes. The run-time support routines are placed in the internal-instruction memory, in the local-memory, or in the shared memory according to their calling frequencies.

The preliminary evaluation results are presented. Corresponding to the hierarchy of PIM/p, two different configuration systems: the network connected system and the common bus connected system, are compared.

The results show that the speedup ratio compared to one PE is nearly equal to the number of PEs for both configuration systems. Hence, the bus traffic is not a performance bottle-neck in PIM/p, and the automatic load-balancing mechanism appropriately distributes loads among PEs within a cluster at the evaluation.

1 Introduction

A parallel inference machine prototype(PIM/p) is now being used. It is tailored to KL1 [Ueda and Chikayama 1990], and includes up to 512 processors. A two-level hierarchical structure is being used in the new system: a processing element and a cluster(Figure 1).

Eight processing elements form a cluster, which communicates with a shared memory through a common bus using snooping cache protocols. The clusters are connected with dual hypercube packet switching networks through network interface co-processors and packet routers. The chassis consists of four clusters. The maximum PIM/p system includes sixteen chassis. A single clock is delivered to all processing elements, maintaining the phase between different chassis.

Some of the features introduced in the PIM/p system are:

- Two level hierarchical structure to allow parallel programming with common memory and to facilitate system expansion with the hypercube network.
- The macro-call instructions which have the advantages of both hard-wired RISC computers and micro-programmable instruction set computers.
- Architectural support for incremental garbage collection *Multiple Reference Bit*(MRB), which reduce memory consumption when the executing parallel logic programming languages such as KL1.
- Each processing element has a local memory, which can reduce bus traffic if the accessed data are placed in the local memory.
- Coherent cache and dedicated cache commands for KL1 parallel execution, which can also reduce common bus traffic.
- Generating the native instruction codes from intermediate KL1-B codes by optimizing compiler with a optimizer.
- The optimizer analyses data-flow for both the tag parts and the data parts independently, which can eliminate unnecessary tag operations.

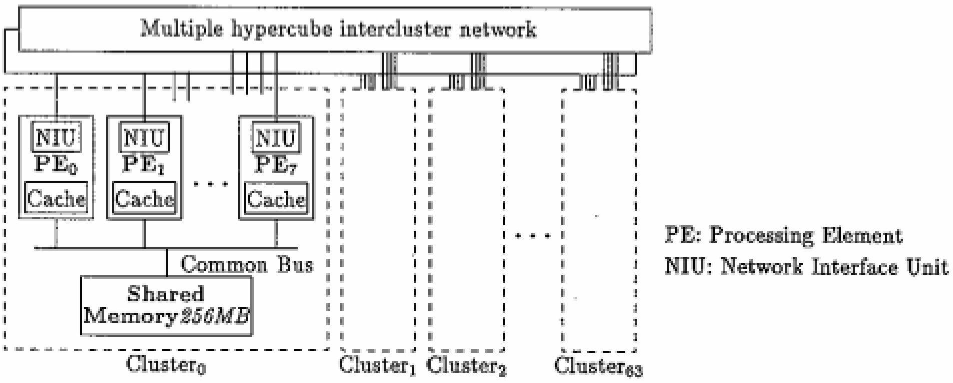

Figure 1: PIM/p system configuration

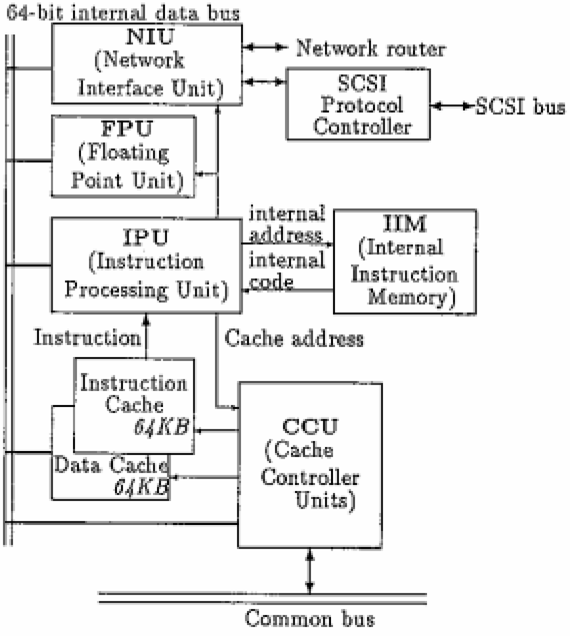

Figure 2: PIM processing element configuration

The *processing Element*(PE) consists of an *Instruction Processing Unit*(IPU), a *Cache Control Unit*(CCU) and network interface unit(NIU). Figure 2 is a schematic diagram of a PE.

In this paper, the hardware architecture and the KL1 processing system are described. In Section 2 to Section 4 we describe IPU, cache and the network system. Then, the run-time support routines for KL1, and the KL1-B compiler code generation and its optimization are described in Section5. Finally in Section 6 a preliminary performance evaluation results are presented.

2 IPU Architecture

The instruction processing unit(IPU) executes RISC-like instructions which have been tailored to KL1 execution. The instruction set has many features which facilitate efficient KL1 program execution. In this section, we describe these features.

2.1 Tagged data and type checking

To execute KL1 programs, a dynamic data type checking mechanism is needed to provide:

- Transparent pointer dereferencing.
- Polymorphic operations for data types.
- Incremental garbage collection support.

Dereference is required at the beginning of most unification operations in KL1. In dereference, a register is first tested to see whether its content is an indirect pointer or not. If it is an indirect pointer, the cell pointed to is fetched into the register and its data type is tested again.

Many operations in KL1 include run-time data type checks even after dereferencing has been completed. Unifications include polymorphic operations for data whose type is not known until run-time.

In addition, incremental garbage collection by MRB is embedded in dereferencing(See Section 2.5 for details).

Therefore, tagged architecture is indispensable for the KL1 processing. In PIM/p, data is represented as 40-bit (8-bit tag + 32-bit data), and the general-purpose register has both a data part and a tag part. The MRB is assigned in one bit of the 8 bit tag.

The tag conditions are specified as bit-wide logical operations between the tag of a register and the 8-bit immediate tag value in the instruction. An instruction can specify the logical operation as *AND*, *OR*, or *XOR* or a negations of one of these.

If an instruction specifies *XOR* as its logical operation, it checks whether the tag of the register matches the immediate value supplied in the instruction. *Xor-mask* operation does this matching under the immediate mask supplied in the instruction, which enables various groups of data types to be specified in a conditional instruction if the data types are appropriately assigned to tag bits (See Section 5.1 for details).

Various hardware flags, like the condition code of ALU operations or hardware exception flags, can be checked as the tags of dedicated registers, so these flags can be examined by a method similar to data type checking.

2.2 Instructions and pipeline execution

The processing element uses an instruction buffer and a four-stage pipeline, **D A T B**, to attempt to issue and complete an instruction. Table 1 shows the pipeline stages in ALU, memory access and branch instructions. All instructions except co-processor instructions are issued in every cycle.

Basic instructions such as ALU operations have three operands, and memory accessing instructions are limited to *load* and *store* type instructions. Pipeline execution tends to make the branch penalty large. In PIM/p, the target instruction starts four clock after the branch instruction starts. To reduce the branch penalty, *delayed branch* instructions are used. These have one delay slot after them.

The *skip* instruction is also useful. This nullifies a subsequent instruction if the skip condition is met. The skip instruction does not cause a pipeline break, so its use results in efficient instruction execution. Figure 3 shows the pipeline stages in conditional branch/delayed-branch/skip instructions.

In the PIM/p pipeline, all instructions write their results at the B stage and ALU or memory write instructions require source operands at the beginning of the B stage. The bypass from the **B** stage can eliminate interlocks. Conditional branch instructions test the condition at the B stage, the bypass also eliminates condition test interlocks. However, when the register is used by address calculation at the **A** stage when the value of the register has just been changed, an interlock may occur even if a bypass from **B** to **A** is prepared. Figure 4 shows this address calculation interlock. The compiler must recognize such interlock conditions and should eliminate them as far as possible.(See section 5.2.3)

```
When conditional branch is taken: condition tested at B
D   A   T   B                    : cond. branch instruction
    D   A   T   canceled         : next external instruction
        D   A   canceled         : 2nd extérnal instruction
            D   canceled         : 3rd external instruction
                D   A   T        : branch target instruction

When delayed branch is used: condition tested at B
D   A   T   B                    : cond. branch instruction
    D   A   T   B                : next external instruction
        D   A   canceled         : 2nd external instruction
            D   canceled         : 3rd external instruction
                D   A   T        : branch target instruction.

When conditional skip is taken: condition tested at B
D   A   T   B                    : cond. skip instruction
    D   A   T   canceled         : next external instruction
        D   A   T   B            : 2nd external instruction
            D   A   T   B        : 3rd external instruction
```

Figure 3: Pipeline stages of conditional branch/skip instructions

```
D   A   T   B                    : register write instruction.
    D   D   D   A   T   B        : inter-lock occurs
                D   A   T        : next instruction
```

Figure 4: Interlock caused by address calculation

2.3 Macro call and internal instructions

A RISC or RISC-like instruction set has advantages in both low hardware design cost and fast execution pipelining. However, naive expansion of KL1-B to low-level RISC instructions produces a very large compiled code. This may cause frequent instruction cache miss-hits and may fill up the common bus band width with instruction feed, especially in tightly-coupled multiprocessors such as a PIM/p cluster. Here, reducing common bus traffic is a most important design issue as is reducing the cache miss-hit ratio. On the other hand, the static code size can be small in a high-level instruction set computer with micro-programs, such as PSI.

To meet both requirements, the processing element of PIM/p has two kinds of instruction streams, *external* and *internal*. External instructions are mostly RISC-like instructions with KL1 tag support[Shinogi et al. 1988]. Internal instructions are fed from internal instruction memory like micro-instructions.

The external instruction set includes macro-call instructions, which first test the data type of a register given as an operand, then invoke programs in the internal instruction memory(IIM) or simply execute the next external instruction, depending on the test result. Every time a macro-call instruction is executed, the corresponding macro-body instruction is fetched from IIM to reduce the calling overhead, but it is not executed unless a macro-call test condition is met (See the **S** and **C** stages of Table 1). Figure 5 shows the pipeline stages of macro-call instructions. A macro-call instruction can be regarded as a light-weight conditional subroutine call or

Table 1: Pipeline stages of ALU, memory access and branch instructions

	ALU operation	*Memory access*	*Branch*
(S)	Set IIM address, *valid only for m-call or internal instructions*		
(C)	Fetch instruction from IIM, *valid only for m-call or internal instructions*		
D	Decode	Decode / Register read for address	Decode / Register read for address
A	—	Memory address calculation	Branch address calculation
T	Register read	Cache tag access	Cache tag access
B	ALU operation / Register write	Cache data access / Register write	Cache data access / Condition test

```
When the condition met: condition test at A
D  A                        : macro-call instruction
   D  canceled              : next external instruction
S  C  D  A  T  B            : first internal instruction
   S  C  D  A  T  B         : 2nd internal instruction

When the condition is not met: condition test at A
D  A                        : macro-call instruction
   D  A  T  B               : next external instruction
      D  A  T  B            : 2nd external instruction
```

Figure 5: Pipeline stages of macro-call instructions

Table 2: The advantages and disadvantages of **B** and **A** condition check

Test stage	*Advantages*	*Disadvantages*
B	No interlock	5τ branch penalty
A	1τ branch penalty	$0/1/2\tau$ interlock

1τ=1 clock cycle

as a high-level instruction with data type checking.

To reduce the overhead of passing parameters from a macro-call instruction to the macro-body, the PIM/p processing element has three *indirect registers*. The indirect registers are pseudo registers whose real register numbers are obtained from the corresponding macro-call instruction parameters.

These mechanisms may appear to be similar to those of conventional micro-programmable computers. Programs stored in IIM are written by system designers into internal instruction memory, like micro-programs. However, the internal instruction set is almost the same as the external instruction set, so a designer can use same development tools to generate both external and internal programs. Therefore, system designers can specify internal or external at the machine-language level, without writing complicated micro-instructions, as in conventional micro-programmable computers.

2.4 Dynamic test stage change

As discussed in the Section 2.3, internal instruction executions require an additional two pipeline stages, S and C, before the D stage, internal conditional branch causes a five clock cycle branch penalty when the branch is taken. In the case of an external branch instruction, target instruction fetch starts at **A** as an operand and the fetch finishes at the **B** stage, thus testing the condition before the **B** stage cannot reduce branch penalty.

However, internal instructions must use the **S** and **C** pipeline stages to fetch the target internal instruction. It cannot not start before the condition test. If the branch condition is determined earlier, say at stage **A**, target fetch can be started earlier. This reduces the branch penalty. However, an early condition test causes interlocking, which is common to memory address calculation, and this will occur even if the branch is not taken. Table 2 shows the advantages and disadvantages of both **B** stage and **A** stage condition tests. Some sample codings show internal conditional branches are often placed just after memory read or ALU operation instructions, and it is hard to insert non-related instructions between them. To minimize pipeline stall, an **A** stage test should be used if the previous instruction does not interlock the condition test, otherwise **B** stage test should be used.

Preparing two sets of branch instructions, a **B** stage test and an **A** stage test, adds instructions to the PIM/p instruction set, because the PIM/p instruction set has many conditional branch instructions for various tag checking.

Without adding instructions, the PIM/p pipeline controller decides between internal conditional branch **A** or **B**[Asato et al. 1991]. When some instructions interlock the test stage **A** of a successive internal conditional branch, the test stage is changed to **B** to avoid interlock, otherwise the test is done at **A** stage. We call this a *dynamic conditional branch test stage change*. If a compiler or a programmer can put two or more instructions between a register write instruction and a conditional branch based on the register, the test is done at the **A** stage.

2.5 MRB support

Incremental garbage collection support is one of the most important issues in parallel inference machines. The PIM/p instruction set includes several instructions for efficient execution of MRB garbage collection[Chikayama and Kimura 1987].

Using the MRB incremental garbage collection, value cells or structures are allocated from free lists, and when those allocated areas are reclaimed, the areas are linked to free lists. To support these free list operations, the *push* and *pop* instructions are used.

The MRB of each pointer and data object has to be maintained in all unification instructions. Especially in dereference, the MRB of the dereferenced result is off if and only if MRBs of both the pointer on a register and the pointed cell are MRB-off. MRB is assigned to one of the eight bit tag data. MRB-on means the bit is 1, MRB-off means 0 respectively. Therefore logical *or* of both the pointer MRB bit and the pointed data MRB bit represents the pointed data's multiple reference status. Dedicated instructions *ReadTagWordMrbor* and *Deref* support this operation. *ReadTagWordMrbor* loads memory data pointed by address register into destination register, accumulates both the address register's MRB and the destination register's MRB that is MRB of the memory data, sets the result status in the destination register. *Deref* is similar to the *ReadTagWordMrbor* instruction, but loads memory data into address register and the old address register value is saved to destination register simultaneously. Therefor, succeeding instructions can examine that the pointed data can be reclaimed or not by testing destination register's MRB bit.

These dedicated instructions can minimize the overhead to adopt MRB incremental garbage collection.

3 Memory Architecture

3.1 Cache and bus protocols

Each PIM/p element processing has two 64K bytes caches for instructions and data. PIM/p uses copyback cache protocols which have been proved effective for reducing common bus traffic in shared-memory multiprocessors. To maintain cache coherence, there are basically two mechanisms, invalidating the modified block and broadcasting the new data to others.

PIM/p uses the invalidation method for the following reasons. To use incremental garbage collection MRB, a reclaimed memory area need not be shared. Next time the area is used it may not be shared with the same processors which previously shared the area. In other, KL1 load distribution is achieved by distributing goal records in a cluster from one processor to another. Usually the distributed goals will not be referred from the source processor. In these cases, the broadcast method will produce unnecessary write commands to the common bus on every write to the newly allocated area or distributed goals. The invalidation method is much more efficient.

PIM/p cache protocol is similar to Illinois protocol. However, PIM/p protocol has the following cache commands optimized for KL1. In normal write operations, a fetch-on-write strategy is used; however, it is not necessary to fetch the contents of shared memory when the block is allocated for a new data structure. That means the old data in the block is completely unnecessary. In KL1, when free lists are recreated after grand garbage collection, the old contents of memory have no meanings. To accomplish this, *Direct_Write* is used.

Direct_write: If cache misses at the block boundary, write data into cache without fetching data from memory.

The following instructions are used for inter-processor communication through a shared memory, for example goal distribution.

Read_Invalidate: When cache misses, fetch the block and invalidate the cache block on other CPUs. This operation guarantees that the block is exclusive unless the other CPU subsequently request the block.

Read_Purge: After the CPU reads a block, it is simply discarded even if it is modified.

Exclusive_read: Same as Read_Invalidate except for the last word in a cache block. When it is used to read the last word in a cache block, it purges the block like *Read_Purge*.

Using these instructions, unnecessary swap-in and swap-out can be avoided by invalidating the sender's cache block after receiver the gets the block, and by purging the receiver's cache block after the receiver reads all data in the block.

Ill-behaved software may cause these instruction to destroy cache coherency. However, these instructions are used only in KL1 processing system, and only systems programmers use them.

There are hardware switches which can change the actions of those special read/write instructions to normal read/write actions. By using these switches, the systems programmer can examine their programs consistency.

3.2 Exclusive control operation

To build a shared-memory parallel processor system, lock and unlock operation are essential guarding critical sections. KL1 requires fine-grain parallel processing. The frequency of locking and unlocking operation needed for shared data is estimated at more than 5% of all memory accesses. Thus these operation must be executed with low overheads by using hardware support. However, locking operations should seldom conflict with each other. It is therefore useful to introduce a hardware lock mechanism which has low overhead when there are no lock conflicts. In PIM/p, the cache block has *exclusive* and *shared* status. When the block is exclusive, it is not owned by other PEs. Hence there is no need to use the common bus. A marker called the lock address register which remembers the block is locked by the CPU. When the CPU locks a block, other CPU cannot get the block data until the block is unlocked by the original CPU. Even when the block is shared, fetching data and invalidating the block before locking is sufficient. The cost is nearly equivalent to the normal write operation.

In KL1 processing, unification requires frequent locking, but the locking time is fairly short. A hardware busy wait scheme is better for lock conflict resolution. If a longer locking time is needed, a software lock can be made by combining *lock*, *read* and *conditional jump* instructions. For KL1, no bus cycles are needed for most of the lock reads hitting exclusive cache blocks.

4 Network Architecture

4.1 Network interface unit

Multiple clusters are connected by a hypercube topology network. At the design stage, we assumed that ten logical reductions require a hundred-bytes packet transfer. The target speed of PIM/p PE will be between 200K LIPS to 500K LIPS. This means 2M to 5M bytes per second network bandwidth is required by each PE. Thus 16M to 40M bytes per second network bandwidth is required to a cluster which contains eight PEs. If this data flows into the common bus, network packet data occupies about 10% to 25% of the total bandwidth of the common bus. Providing a network interface to each processing element reduces such common bus traffic.

Each cluster has 8 PEs, and each PE has a network interface co-processor called a *network interface unit* (NIU). By attaching a NIU to each PE, a PE can send to or receive from a packet without using the common bus. The NIU performs the following functions:

- Builds a packet into the NIU's packet memory, and sends it to the network router(RTR).
- Receives a packet from the RTR, stores it to the packet memory. and signals the arrival of a packet to IPU.
- Communicates to a SCSI bus driver chip which connects to PIM/p front-end processors(FEPs) or disks.

All these actions are controlled by the IPU's co-processor instructions.

To build a packet, the IPU first makes a header which contains the packet destination and mode for broadcasting. It then building a packet body by executing co-processor write instructions, which packs data one, two, or four bytes at a time. Finally the IPU puts a end of packet marker to send the packet to RTR. A whole packet of data is stored in packet memory before sending it, to minimize RTR busy time. The send and receive packet memories are both 16K bytes long.

Each cluster has four SCSI ports which are connected to the PEs. Two have non-differential SCSI interface ports, and the other two have differential SCSI interface ports. The differential SCSI interface is able to extend the interface cable up to twenty five meters. It is used to connect SCSI disks which need not be placed beside the cluster. The PIM/p FEP is connected to a non-differential interface, and various other SCSI devices, such as an ether-net transceiver, can be connected through the SCSI bus. This extends PIM/p's application domain.

4.2 Inter-cluster network connection

While the NIU sends and receives packets, the network packet router(RTR) actually delivers packets. Each RTR connects four NIUs and up to six other RTRs to build a sixth order hypercube network topology. Thus each cluster has two RTRs which construct two independent hypercube networks to improve the total network throughput. The RTR can connect a maximum of sixty-four clusters(512 PEs).

RTR uses the wormhole routing method to reduce traveling time when the network is not so busy, to avoid packet length restrictions caused by RTR packet buffer limitation. Between RTRs data is transferred at system clock rate. RTR has approximately 1K bytes of packet buffer for every output port, in order to reduce network congestion. The static routing method is used and deadlocks are avoided by the routing method. Broadcasting to the sub-cube is available. This can be used when the system is at the initial program stage.

In the PIM/p system, one chassis contains four clusters. The maximum 512PE PIM/p system is sixteen chassis. Building for such a large system can be problematic. Transferring data between these chassis by synchronous-phase matched clock is impossible, because the system occupies an area of about sixteen meters square. This means that the traveling time of data is about one system clock tick. Introducing another hierarchy between inner-chassis communication and inter-chassis communication complicates the distribution strategies of the KL1 processing systems. This should be avoided.

One of main feature of RTR is the interconnection between PIM/p chassis. To attain a transfer rate equal to system clock rate for both inner-chassis and inter-chassis data, RTR uses a data synchronization mechanism for inter-chassis connections. This makes the inter-chassis connection transfer rate equal to the inner-chassis transfer rate, with little increase in data traveling time. This simplifies the cluster hierarchy.

5 The Kl1 Language Processing System for PIM/p

The KL1 language processing system for PIM/p is designed on the basis of the VPIM [Hirata et al. 1992]; it is the common specifications of the KL1 language processing system on the two level hierarchical multi-processor system. Most specifications of VPIM are used for PIM/p with no changes. Some modification, however, were applied to exploit the PIM/p hardware efficiently.

The KL1 language processing system is implemented as the KL1 compiler and the run-time support routines. The KL1 program must be compiled into PIM/p native machine code when it is executed on PIM/p. The KL1 compiler for PIM/p consists of three passes — the compiler to the intermediate code, the native machine code generator and the optimizer. Compiled KL1 programs may call some run-time support routines as circumstances demand. The run-time support routines are classified into three groups, which correspond to PIM/p memory architecture.

5.1 Changes for PIM/p

There are some changes from VPIM to PIM/p. These were applied to exploit the PIM/p hardware efficiently.

(1) Data Structure

The basic KL1 data are realized by tagged words; each of them consists of a 8-bit tag part and a 32-bit value part, and all KL1 data are realized by tagged words in VPIM. The memory of PIM/p consists of 64-bit width words. Tagged words are placed in aligned 64-bit width words in the PIM/p memory system [Goto et al. 1988]. Although KL1 data density will be low in this scheme, this will not cause performance degradation.

The PIM/p instruction processing unit can access the memory not only in the unit of tagged data, but also in the 8-bit, 16-bit, 32-bit and 64-bit units. A string — an array of integers can, therefore, be realized using 64-bit width words, as shown in figure 6. A *module* which holds KL1 compiled code, is also realized under the same scheme. Since PIMOS [Chikayama et al. 1988] uses many string data and *module* data, this scheme can promote efficiency of memory using.

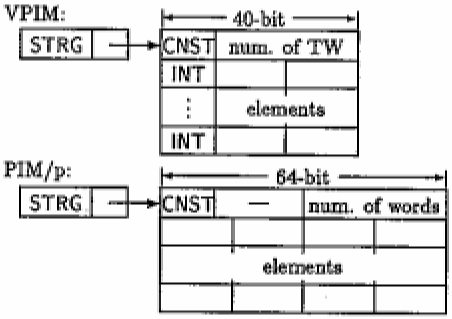

Figure 6: String data of VPIM and PIM/p

(2) Data Type Checking

The PIM/p instruction processing unit has special instructions for data type checking: *JumpXorUnderMask* and *JumpNotXorUnderMask*. These have the following functions:

if(tag_of(Reg)&Mask = Const) goto Label;
and
if(tag_of(Reg)&Mask ≠ Const) goto Label;

These functions can test not only if the data type is correctly specified, but also if the data type group is correctly specified, since the bit assignment of tag field is designed effectively.

The KL1 language processing system uses 44 kinds of data types; these can be expressed in 6 bits. The tag part, however, is 7-bit width except MRB. We use 7 bits in a tag part to express data type; data types are assigned sparsely in order to check data type group easily by *JumpXorUnderMask* or *JumpNotXorUnderMask*. There are the following data type groups:

- Atomic — atom or integer.
- Vector — null vector, short vector or long vector.
- Short Vector — vector containing 1–8 elements.
- Undefined — variable in some conditions.

These data type groups are often checked in KL1 execution, and this assignment can reduce execution costs.

5.2 Compiler

The KL1 program must be compiled into PIM/p native machine code when it is executed on PIM/p. The KL1 compiler for PIM/p consists of three passes — the compiler to the intermediate code, the native machine code generator and the optimizer. In the first pass, the KL1 program is compiled into intermediate code; its instruction set is called KL1-B. The native machine code generator expands intermediate code into PIM/p native machine code. The optimizer improve the expanded code.

5.2.1 Intermediate Code

In the first pass of the KL1 compiler, the KL1 program is compiled into intermediate code; its instruction set is called KL1-B. It is designed as the instruction set for the *abstract KL1 machine* [Kimura and Chikayama 1987] and interfaces between the KL1 language and the PIM hardware, just as the Warren Abstract Machine [Warren 1983] does for Prolog. The KL1-B for PIM is extended from KL1-B for Multi-PSI to exploit the PIM hardware efficiently.

KL1-B contains passive unification instructions, active unification instructions, argument/element preparation instructions, incremental garbage collection instructions and goal manipulation instructions. These specifications are identical with VPIM [Hirata et al. 1992].

5.2.2 Native Machine Code Generator

The intermediate code, which consists of KL1-B instructions, is expanded into native machine code according to the *template*; the *template* is a set of rules governing translation from KL1-B instructions to native machine instructions. These rules are defined according to the following principles:

- Use the special instructions for KL1 effectively.
- Don't jump in the main pass.
- Minimize the pipeline break ratio.
- Maximize the hit ratio of the instruction cache.

The translating rules are classified into the following 3 groups according to the properties of the KL1-B instructions.

(1) Expand to In-Line Code

These KL1-B instructions which can always be realized by a few native machine instructions are translated accordingly. Consider the following examples:

```
load  Rgp,Pos,Reg
  → ReadTagWordShortOffset    Reg,Pos*8+40(Rgp)
read  Rsp,Pos,Reg
  → ReadTagWordMrbOr          Reg,Pos*8(Rsp)
is_vector  Reg,Lab
  → JumpNotXorUnderMask       Reg,VG,Lab,VGM
put_integer  Const,Reg
  → MoveTagWordWithTag        Rzero,Reg,INT
      (Const = 0)
  → AddImmediate              Reg,Rzero,Const
    MoveTagWordWithTag        Reg,Reg,INT
      (0 < Const < 256)
  → LoadImmediate             Reg,Const
    MoveTagWordWithTag        Reg,Reg,INT
      (Const ≥ 256 or Const < 0)
```

Load is translated into a single native machine instruction. In this sample, *Pos*, the position specifying an argument, is adjusted to the offset in a byte unit.

Read is not a simple read instruction; it must maintain the MRB. PIM/p, however, has a special instruction for this use. *Read* can be realized by a single native machine instruction: *ReadTagWordMrbOr.*

Is_vector tests if the data type group of *Reg* is a vector group. This is translated into a single native machine instruction: *JumpNotXorUnderMask.*

Put_integer has three translation rules from which is selected according to the value of *Const*, in order to generate fast, concise code. These translated codes take 1clock-cycle/4bytes, 2clock-cycles/8bytes and 2clock-cycles/10bytes respectively.

(2) Expand to Conditional Subroutine Call

The KL1-B instructions whose main pass can be realized by a few native machine instructions are translated into these instructions, together with the instructions calling a subroutine conditionally. The subroutines are classified into two groups; the *macro library* and the *roundabout routines.*

The *macro library* is a set of the run-time support routines and called by the *macro call* instruction. These routines realize common functions in executing KL1, and are shared with all compiled codes (See section 5.3). Consider the following examples:

```
reuse_vector  Arity,Reg
  → MacroCallAnd    Reg,MRB,Arity,m_AllocVector
```

Reuse_vector does nothing when the MRB of the vector pointer on the register is not marked. It can, therefore, be translated into a single *conditional macro call* instruction. When the MRB of the pointer is marked, *reuse_vector* allocates a new vector; this allocation is done in the *macro library.*

The *roundabout routine* is placed in the compiled code of the KL1 program. It realizes a local function, and is used from the compiled code of a single KL1-B instruction. Consider the following example:

```
reuse_vector_with_elements  3,Reg,{1,0,1}
  → JumpAnd                        Reg,MRB,LC001
   LR001:
          ⋮
   LC001:
     MacroCall               Rwork1,0,Arity,m_AllocVector
     ReadTagWordMrbOr              Rwork2,0(Reg)
     WriteTagWordShortOffset       Rwork2,0(Rwork1)
     ReadTagWordMrbOr              Rwork2,16(Reg)
     WriteTagWordShortOffset       Rwork2,16(Rwork1)
     JumpDelayed                   LR001
     MoveTagWord                   Rwork1,Reg
```

Reuse_vector_with_elements is translated into a single *JumpAnd* instruction as a main pass, and some additional instructions as the *roundabout routine*. In KL1 applications, the MRB of the structure pointer is often unmarked, and *roundabout routine* is not executed. This *roundabout routine* is changeable according to the third arguments of the KL1-B instruction. It cannot, therefore, be shared with some KL1-B instructions.

(3) Expand to Subroutine Call

The KL1-B instructions which always execute complicated functions are translated into the subroutine call instruction or the *macro call* instruction. The processing of complicated functions are executed by run-time support routines. Most KL1-B instructions for active unification and body built-in predicates are translated using this rule. This is because the calling cost is low compared to the cost of executing complicated functions, and the size of the compiled code can be minimized.

5.2.3 Optimizer

The compiler for PIM/p supports the optimization of the expanded code; the expanded code is the native machine code translated from the intermediate code according to the *template*. Since expansion according to the *template* is applied to each KL1-B instruction separately, some redundant instructions may be generated, and the order of instructions is not refined. Optimization is applied to the expanded instructions as a group, and these instructions are removed. Two optimization techniques are introduced.

(1) Optimization by Data Flow Analysis

The optimizer analyzes data flow among the instructions in the expanded code. It then trims some redundant instructions and merges some instructions into a single instruction; for example:

- The optimizer trims the instruction which puts a datum onto a register, even if the datum is not used later.
- The optimizer generates an instruction which calculates with a constant datum, instead of an instruction which puts the constant onto a register and an instruction which calculates with the datum on the register.

In this optimization, the data flow analysis is applied separately to the tag part and the value part of a datum. This is because the KL1-B always treats a datum as a set of the tag part and the value part, while some native machine instructions disregard the value part.

The sample code is shown as follows; this is the compiled code of a guard built-in predicate: *add(X,1,Y)*.

- intermediate code:

```
put_integer    1, R2
integer_add    R1, R2, R3, fail
```

- native machine code (not optimized):

```
AddImmediate           R2, Rzero, 1
MoveTagWordWithTag     R2, R2, INT
Add                    R3, R1, R2
JumpAnd                CCR, CC_V, fail
```

- native machine code (optimization #1):

```
AddImmediate           R2, Rzero, 1
Add                    R3, R1, R2
JumpAnd                CCR, CC_V, fail
```

- native machine code (optimization #2):

```
AddImmediate           R3, R1, 1
JumpAnd                CCR, CC_V, fail
```

There are two KL1-B instructions, and each of them is expanded into two native machine instructions. In the unoptimized code, the *Add* instruction uses *R2* as the input, but disregards the value part; therefore the *MoveTagWordWithTag* instruction has no effect and can be removed (optimization #1). Additionally, *AddImmediate R2,Rzero,1* and *Add R3,R1,R2* can be merged into a single native machine instruction: *AddImmediate R3,R1,1*. In this sample, optimized code takes 2clock-cycles/10bytes while unoptimized code takes 4clock-cycles/18bytes.

(2) Pipeline Optimize

The processing element for PIM/p uses a four-stage pipeline. In expanded code, the dependencies between instructions which have been expanded from different KL1-B instructions, are disregarded, and delayed branch instructions are not used as often. The optimizer rearranges the order in which instructions are executed, to ensure smooth pipeline processing.

In KL1 execution, pointer operations — pointer readings and address calculations are often done while pointer operations may cause interlocks. This optimization, there fore, is very effective.

5.3 Run-time Support Routines

The run-time support routines are called from the compiled KL1 program in order to execute complicated functions. They are divided into three groups corresponding to PIM/p memory architecture (Figure 7).

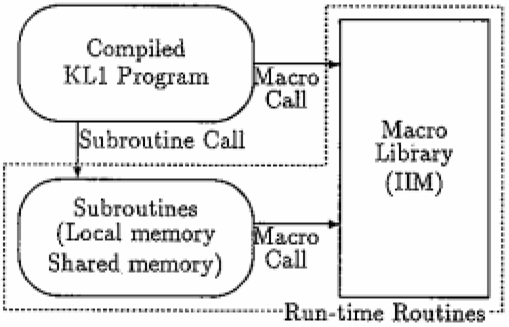

Figure 7: Run-time support routines

(1) Macro Library

The *macro library* is called using *macro call* instructions. This is a kind of subroutine library, but is stored in the *internal instruction memory* (IIM) of IPC, like microprograms. There are no instruction cache misses.

The characteristics of *macro call* instructions are as follows:

- In a *macro call* instruction, a *tag conditional branch*, applied to a run-time KL1 data type check, is carried out in one instruction step.
- Argument registers or short (8-bit) immediate values are specified in the *macro call* instruction, so the operands of a *macro call* can be efficiently passed to its *macro library* function.

The IIM can store 8K-step instructions. We implement the most frequently used functions, for example, the dereference and unification functions, in the *macro library*.

(2) Frequently-used Libraries

Other frequently-used libraries are stored in local memory. The cost of instruction fetches in local memory is small, because it doesn't use the common bus.

Functions for the suspend/resume processes for KL1 goals and the copying GC routines, are implemented in these libraries.

(3) Occasionally-used Libraries

. Occasionally-used libraries are stored in shared memory. Access speed for shared memory is slower than that for local memory or IIM, but the storage is so large that we can implement complicated libraries in this memory.

We implement most of the body *built-in* predicates, the network control routines and the *shoen* (meta-function) control routines for these libraries [Hirata et al. 1992].

Table 3: Speedups for Pentomino

Number of PEs	1	2	4	8
Memory shared system	1.00	1.96	3.86	7.50
Network connected system		1.93	3.80	7.28

6 Evaluation

We used Pentomino as a benchmark program and executed it on two system configurations — multi-PE×1CL and 1PE×multi-CL. The multi-PE×1CL configuration represents the memory shared multi-PE system, and the 1PE×multi-CL configuration represents the network connected multi-PE system.

Pentomino is a program to find out all solutions of a 5×8 packing piece puzzle; packing a 5×8 rectangular box by ten various shaped pieces, each is made up of four unit squares. The program does an exhaustive search of an OR-tree of possible pieces elements.

The benchmark program for the network connected multi-PE system contains the *multi-level load balancing* [Furuichi et al. 1990] code which requires the optimization for the network configuration. However, the program for the memory shared multi-PE system does not contains load balancing code.

On the memory shared multi-PE system, the load balancing in a cluster is executed automatically with a KL1 goal as a unit. Each PE has two goal pools, one is local for the PE, the other is public; it is accessible from other PEs. If a PE has many goals in its local pool, it moves some goals into its public pool. The goals in the public pool might be executed by any PEs in the cluster.

Table 3 shows that the speedup ratio according to the number of PEs is nearly equal to the number of PEs for two system configurations. The automatic load balancing mechanism of the memory shared multi-PE system works as efficiently as the optimized load balancing code for the network connected multi-PE system.

7 Conclusion

PIM/p has up to 512 PEs using two level hardware structures. Two level hierarchical structure allows parallel programming with common memory and facilitates system expansion with the hypercube network. On the two level hierarchical structure system, programmers do not think about load balancing inner cluster and write only the load balancing code for clusters.

The special cache control instructions and the macro-call mechanism reduce the common bus traffic, which may become the performance bottle-neck for shared memory multi-PE systems. The evaluation result shows that the speedup is linear upto 8 PEs in a cluster. The common bus traffic, therefore, does not become the performance bottle-neck.

The macro-call mechanism reduces the costs of type checking and the overhead of passing parameters. Using this mechanism, it becomes easier to implement the KL1 language processing system.

Acknowledgment

We wish to thank all of the PIM research members both at ICOT, at Fujitsu Social Science Laboratory Ltd. and at Fujitsu Limited. Especially we thank ICOT researchers, Dr. K. Hirata and Mr. A. Imai for their useful comments. We also wish to thank Mr. A. Shinagawa and Mr. H. Miyake of Fujitsu Limited for their helpful support in developing softwares. Finally, we would like to thank the director of ICOT research center, Dr. K. Fuchi, the manager of research department, Dr. S. Uchida, the chief of first research laboratory, Dr. K. Taki, the general manager of processor division in Fujitsu Laboratories Ltd, Mr. J. Tanahashi, and the general manager of advanced information systems division in Fujitsu Laboratories Ltd, Mr. H. Hayashi, for their valuable suggestions and guidance.

References

[Asato et al. 1991] A.Asato, M.Kimura, T.Shinogi, A.Hattori. A Pipeline Control Method of PIM/p. In *Proceedings of 43rd Anual convention IPS Japan*, 1991 (In Japanese).

[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*, pp.276–293, 1987.

[Chikayama et al. 1988] T.Chikayama, H.Sato and T.Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.230–251, 1988.

[Furuichi et al. 1990] M.Furuichi, K.Taki and N.Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proceedings of 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[Goto et al. 1988] A.Goto, M.Sato, K.Nakajima, K.Taki and A.Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.208–229, 1988.

[Goto et al. 1990] A.Goto, T Shinogi, T.Chikayama, K.Kumon and A.Hattori. Processor Element Architecture for a Parallel Inference Machine, PIM/p. In *Journal of Information Processing*, pp.174-182, Vol.13, No.2, 1990.

[Hirata et al. 1992] K.Hirata, R.Yamamoto, A.Imai, H.Kawai, K.Hirano, T.Takagi, K.Taki, A.Nakase and K.Rokusawa. Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1992.

[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*, pp.468–477, 1987.

[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 *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988.

[Taki 1992] K.Taki. Parallel Inference Machine PIM. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1992.

[Ueda and Chikayama 1990] K.Ueda and T.Chikayama. Design of the Kernal Language for the Parallel Inference Machine. *The Computer Journal*, (33)6, 1990, pp.494–500.

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