

Architecture and Implementation of PIM/m

Hiroshi Nakashima[†] Katsuto Nakajima*
Seiichi Kondo[†] Yasutaka Takeda*
Yū Inamura[†] Satoshi Onishi[†]
Kanae Masuda*

* Mitsubishi Electric Corporation

5-1-1 Ofuna, Kamakura, Kanagawa 247, Japan

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

Abstract

In the FGCS project, we have developed a parallel inference machine, PIM/m, as one of the final products of the project. PIM/m has up to 256 processor elements (PEs) connected by a 16×16 mesh network, while its predecessor, Multi-PSI/v2, has 64 PEs. A PE has three custom VLSI chips, one of which is a pipelined microprocessor having special mechanisms for KL1 execution, such as pipelined data typing and dereference.

As for the KL1 implementation on PIM/m, we took much care of garbage collection and introduced various techniques, such as incremental reclamation of local and remote garbage. Especially, a hardware mechanism to support the local garbage collection greatly contributes to reducing the overhead and achieving high peak performance, 615 KLIPS in *append* on single processor.

Sustained performance of single processor is also improved, and is approximately twice as high as that of Multi-PSI/v2. This improvement and the enlargement of the system scale cooperatively enhance the total system performance, and make PIM/m 5 to 10 times as fast as Multi-PSI/v2.

1 Introduction

Several parallel inference machines have been developed in the Japanese Fifth Generation Computer Systems (FGCS) project. As a part of this activity, we have developed three parallel machines. The first machine, Multi-PSI/v1 [Masuda *et al.* 1988, Taki 1988], was an experimental version and was completed in 1986. It has 6 processor elements (PEs) each of which is our first sequential inference machine, PSI-I [Taki *et al.* 1984], and has a software interpreter for the machine language KL1 which is an extended version of flat GHC [Ueda 1985]. Though the machine scale was small and the performance was not very high, the development of Multi-PSI/v1 gave us valuable experimental knowledge of the distributed implementation of KL1 [Ichiyoshi *et al.* 1987].

The second machine is Multi-PSI/v2 [Takeda *et al.* 1988, Uchida *et al.* 1988], which contains 64 PEs connected by two-dimensional mesh network. Each PE consists of PSI-II's CPU kernel [Nakashima and Nakajima 1987], a network controller, and an 80 MB local memory. KL1 programs are compiled to WAM-like machine instructions for KL1 [Kimura and Chikayama 1987] executed by a microprogrammed emulator. The large machine scale and high performance, owing to the improvement of the processor architecture and implementation technology, make Multi-PSI/v2 the first practical parallel inference machine. Its operating system, PIMOS [Chikayama *et al.* 1988], also greatly contributes to its availability by providing highly sophisticated environment for parallel programming. Thus, many KL1 programs for various application areas have been developed on it since its first model was shipped in 1988 [ICOT 1990]. These programs and many users of 15 systems prove the efficiency and practicality of Multi-PSI/v2.

Then, we have just finished the development of our final machine, PIM/m. It inherits many architectural features, such as the mesh network and KL1 execution mechanism, from Multi-PSI/v2. The performance, however, is greatly improved by drastically modifying PE architecture and increasing the number of PEs to 256.

In this paper, the hardware architecture of PIM/m and the KL1 implementation on it are described. Section 2 shows the system configuration, and the architecture of PE and its processing unit. Section 3 describes several topics about the KL1 implementation emphasizing the relation with garbage collection. Section 4 presents preliminary performance evaluation results and analysis on them.

2 Hardware Architecture

2.1 System Configuration

Figure 1 shows the overview of PIM/m 256 processor system. PIM/m consists of up to 8 cabinets, each of which contains 32 PEs connected to form an 8×4 mesh

[†]hiroshi@isl.melco.co.jp

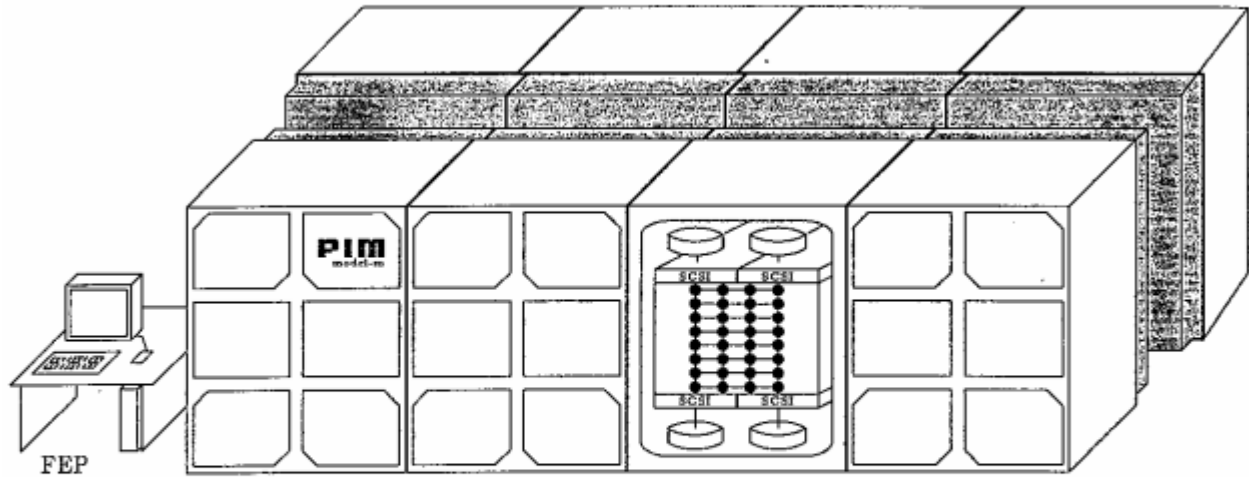


Figure 1: Overview of PIM/m

sub-network. This sub-network is embedded in a larger network, up to 16×16 , by channels connecting adjacent cabinets. Thus we can provide various size systems, from 32 to 256 PEs.

A cabinet also contains four 670 MB disks, which make a 256 PE system have huge disk space, larger than 20 GB. This huge capacity should be enough for applications such as knowledge base and genetic information analysis. Each disk is coupled with a PE by SCSI bus, which is also used to connect other special I/O devices, other PIM systems, and/or front end processors (FEP).

The FEP is a high performance AI workstation, PSI/UX [Nakashima *et al.* 1990]. It has a special attachment processor to execute a sequential logic programming language ESP [Chikayama 1984]. Since the CPU kernel of FEP is that of PIM/m's PE, FEP is also capable to execute KL1 in single processor environment or simulated multiprocessor environment. Therefore, programmers use FEP not only as an interactive I/O system, but also as a convenient debugging workbench.

2.2 Processor Element

Each PE has three VLSI chips, PU (Processing Unit), CU (Cache Unit) and NU (Network Control Unit), as shown in Figure 2. These chips and other peripheral chips including a floating point processor are installed on one printed circuit board. The other board carries a 16 M-word (80 MB) local memory constructed from 4 M-bit DRAM chips. This two board configuration of PE is much smaller than that in Multi-PSI/v2, eight boards, and makes it possible to increase number of PEs from 64 to 256, owing to the advanced VLSI technology. The machine cycle is 65 ns, which has also been improved

from 200 ns of Multi-PSI/v2.

PU is a 40-bit pipelined microprocessor which executes KL1 (and ESP in FEP) under the control of a microprogram stored in 32 K-word writable control store. The architecture of PU is described in 2.3 and 2.4. CU contains a 1 K-word (5 KB) instruction cache and a 4 K-word (20 KB) data cache.

NU performs switching of message packets transferred through the mesh network, using so-called *Worm-Hole Routing* mechanism. As shown in Figure 3, the network of PIM/m consists of full duplex channels connecting adjacent PEs. That is, a pair of adjacent PEs may simultaneously transmit message packets to each other. Moreover, a message packet passing through a PE does not disturb the KL1 execution on the PE, nor collide with others unless they have the same direction.

The network is invested with these properties by the

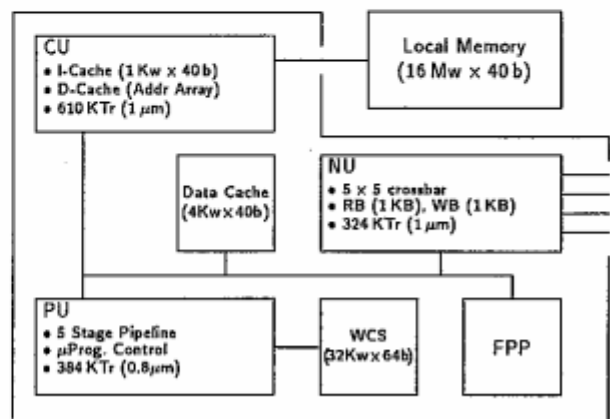


Figure 2: Processor Element

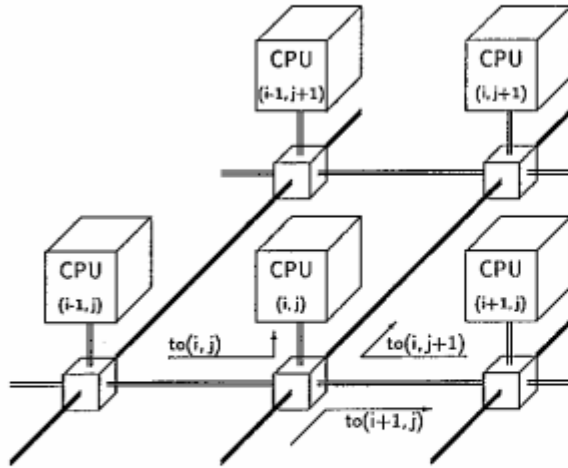


Figure 3: Network Configuration

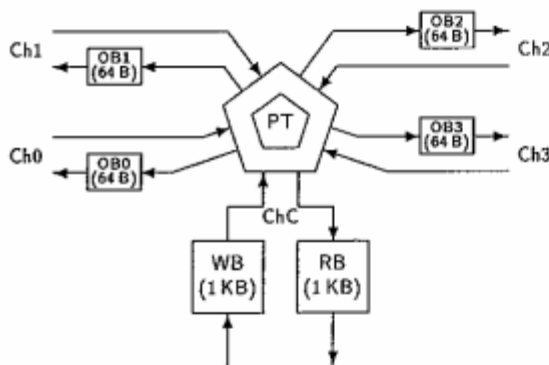


Figure 4: Network Control Unit (NU)

architecture of NU shown in Figure 4. NU has a 5×5 crossbar to switch four input/output channel pairs for adjacent PEs (Ch0-3) and a pair for CPU (ChC). These channels carry a 9-bit packet byte, which consists of 8-bit data field and a *mark* to indicate the header and trailer of a packet.

Switching is performed by looking up a RAM table named *path table* (PT). The address of the table is provided from the packet header which specifies the destination PE number of the packet. Each entry of the table contains a 2-bit code indicating the direction of the packet, going straight, turning left/right, or arriving at its destination. This mechanism gives us much flexibility for routing, system reconfiguration, and physical interconnection of PEs. As the path table has independent read ports for each input channels, collision of packets does not occur even in switching phase.

Once the connection of an input/output channel pair is established, NU transmits a packet byte per four machine cycles regardless the physical location of the adjacent PE. This feature is owing to a sophisticated asynchronous communication mechanism using FIFO *output buffers* (OB0-3). In this mechanism, a sender PE does not wait for acknowledgment from the receiver for a packet byte. Instead it cares about the caution from the receiver saying that the output buffer on the next path will soon be full. Since the caution is raised before the buffer is really full taking physical line delay into account, packet bytes never overrun. The output buffers also contribute to reducing the probability of network choking.

The channel pair for CPU has two FIFO buffers, a *read buffer* (RB) and a *write buffer* (WB). The read buffer acts as an output buffer for the packets directed to the PE itself. Its size 1 KB, however, is much greater than that of output buffers, 64 B, in order to hold a whole message packet. When the tail of a packet written into the read buffer, an interrupt raises to tell CPU that the packet arrives. The write buffer, whose size is also 1 KB, starts transmission of a packet when its tail is written, in order to avoid that the packet is chopped. Both buffers also have the capability to compose/decompose a 40-bit word from/into packet bytes.

2.3 Processing Unit (PU)

Figure 5 shows the configuration of the processing unit, PU [Nakashima *et al.* 1990, Machida *et al.* 1991]. PU executes WAM-like instructions for KLI, named KLI-B [Kimura and Chikayama 1987, Warren 1983], with the following registers.

- A_n/X_n .. Argument and temporary registers.
- PC Program counter.
- AP Alternate clause pointer.
- CGP Current goal record pointer.
- GSP Goal stack pointer.
- SST Suspension stack top.
- HP Heap pointer.
- SP Structure pointer.
- FVP Free variable cell pointer.
- FLP Free list cell pointer.
- FGP Free goal record pointer.

A_n/X_n are implemented as a register file. The other register file, WR, contains the control registers shown above, except for PC and SP which are hardware counters. Each register is 40 bit width, including 8 bit tag for data type

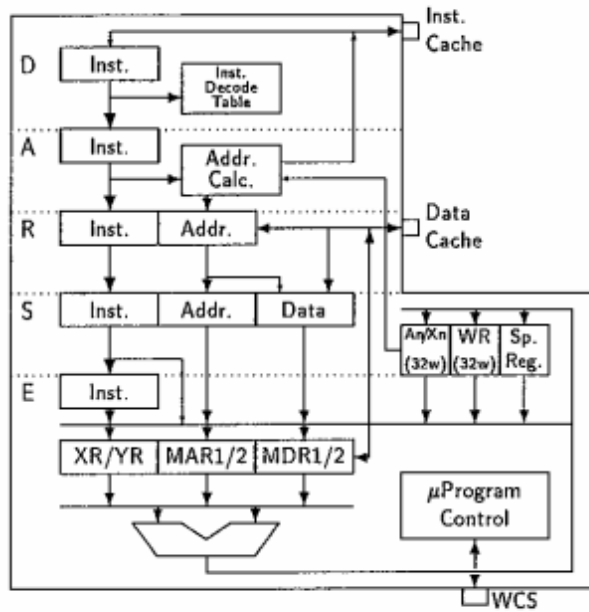


Figure 5: Processing Unit (PU)

representation and incremental/ordinary garbage collection. The tag bit for incremental garbage collection is called *Multiple Reference Bit* (MRB) described in 3.1.

PU has five pipeline stages, D, A, R, S and E.

The D (Decode) stage has a RAM table for instruction decode. Each entry of the table contains the start address of the microprogram routine for an instruction, and the *nano-code* to control the following stages. This RAM decoder makes it easy to develop the microprogram.

The A (Address Calculation) stage calculates the operand address by adding two of following resources, according to the nano-code.

- An operand field of the instruction.
- Program counter, PC.
- A_n/X_n specified by an operand field.
- Current goal pointer, CGP, to get a location of a goal argument.

The A stage also controls instruction fetch, including conditional and unconditional branch operations.

The R (Read Data) stage fetches an operand from data cache using the calculated address, if necessary. The S (Setup) stage selects three operands from the following resources and transfers them to the E (Execution) stage, according to the nano-code.

- An operand field of the instruction.
- The operand fetched by the R stage and its address.

- A_n/X_n specified by an operand field.
- Control registers in WR
- Structure pointer, SP.

In conventional pipelined processors, the operand setup operation is performed by the stage like R. PU, however, has an additional special stage, S, for the operation. The reason for introducing the S stage is that it is required for the pipelined data typing and dereference, as discussed later.

The E stage has two pipelined phases controlled by microinstructions. The first phase contains A_n/X_n , WR, and special registers including PC and SP. This phase is shared by the S and E stages for the operand setup. The second phase has two temporary registers (XR/YR), two memory address registers (MAR1/2), and two memory data registers (MDR1/2). Two of those registers are input to ALU, and the result is written into registers in the first and/or second phase. ALU operation and tag manipulation including turning on/off MRB are performed in parallel.

2.4 Data Typing and Dereference

Data typing and dereference are very important for efficient implementation of logic programming languages. Both data typing and dereference are performed by checking the tag of data and changing the control flow according to the result. PU has powerful mechanisms, including the pipelined data typing and dereference, for these operations.

The E stage has the following microprogram operations for tag checking.

- (1) Two-way conditional jump. The jump condition is obtained by comparing the tag of a register with an immediate value or the tag of another register.
- (2) Three-way jump. The tag of MDR1 or MDR2 is compared with an immediate value and *reference* tag.
- (3) Multi-way jump. A RAM table, which contains jump offsets, is looked up by the tag of MDR1 or MDR2.

These operations requires two machine cycles. The first cycle makes the jump condition or offset, and the second generates the jump address and fetches the microinstruction.

The pipelined data typing and dereference, which are most unique features, mainly depend on the S stage. The S stage has the following three functions for data typing.

- (1) Modify the microprogram entry address comparing the tag of the operand fetched by the R stage with an immediate value.

- (2) Set up the offset of a multi-way jump, which can be performed by the first microinstruction, looking up the RAM table by the tag of the operand fetched by the R stage.
- (3) Set up the two-way jump condition, which can be examined by the first microinstruction, comparing the tag of an operand transferred to the E stage with an immediate value.

The first two functions require the special stage between the R and E stages.

The S stage also performs dereference. When the dereference from A_n/X_n is ordered, the R stage fetches the operand if the A_n/X_n contains reference pointer, while it always fetches the operand in the case of the dereference from memory. In both cases, the S stage examines the tag of fetched data, and repeatedly reads memory until a non-reference data is obtained. The state of the reference path indicated by MRB of each reference pointer is also examined, as described in 3.1.

3 Implementation

Since logic programming languages don't have destructive assignment, manipulating a data structure often makes a copy of the data leaving its old version as a garbage. In Prolog, garbage data cells may be reclaimed by intentional backtrack with side effect operations. In KL1, however, this technique cannot be used because *deep* backtrack causes the failure of the entire program. Thus, garbage reclamation has to be performed only by the run-time system.

In the KL1 implementation on PIM/m, therefore, we took much care of garbage collection and its efficiency. For the reclamation of *local* garbages, an incremental garbage collection using *Multiple Reference Bit* (MRB) is introduced. *Remote* garbages, which was once pointed from PEs other than its home, are also reclaimed incrementally by a sophisticated reference counting mechanism for reducing the number of inter-PE messages, called *Weighted Export Counting* (WEC).

This section describes the implementation of KL1, emphasizing these garbage collection mechanisms and related techniques to reduce memory space and number of messages.

3.1 Local Incremental Garbage Collection

Concurrent processes in KL1 communicate each other through shared logical variables. Typically, a pair of concurrent processes, a producer and a consumer, has its own logical variable in which the producer puts some data by an *active* (or *body*) unification. The consumer

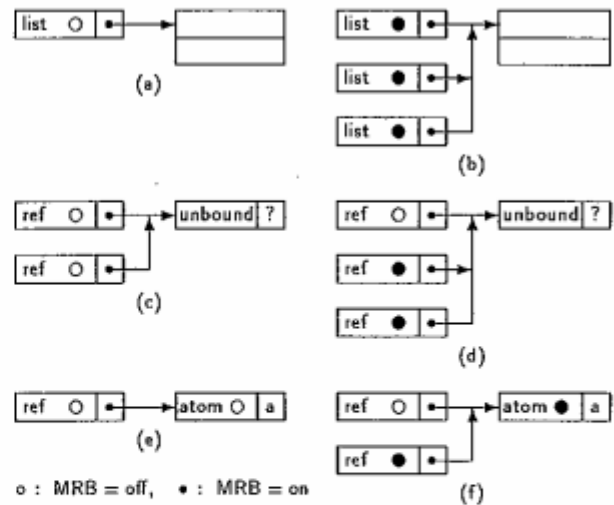


Figure 6: Multiple Reference Bit (MRB)

will be activated by binding the variable and read its contents by a *passive* (or *guard*) unification. Because the variable cell is shared only by the producer/consumer pair, it will become garbage after the consumer gets its contents. Moreover, a structured data unified with the variable may also become garbage after the consumer decomposes it.

The *Multiple Reference Bit* is introduced in order to reclaim these garbages [Chikayama and Kimura 1987]. MRB is a one-bit reference counter attached to both pointers and objects. As the counter for a pointer, MRB is turned on (overflowed) if the pointer is duplicated, as shown in Figure 6(a) and (b). That is, a pointer with MRB on might refer to an object together with other pointers. In other words, an object directed by a pointer with MRB off can be reclaimed as a garbage after the (passive) unification is performed through the pointer.

This rule, however, has an exception for unbound variables each of which can have two reference pointers with MRB off, for a producer and a consumer (Figure 6(c)). After the producer unifies the variable with some data and loses its reference path to the variable, the path from the consumer to the data is left alone as the rule requires.

This exception leads to the other aspect of MRB, counter for an object. As shown in Figure 6(d), an unbound variable might have a pointer with MRB off and two or more pointers with MRB on. If the variable is unified with a data through the pointer with MRB on, the data has a pointer with MRB off, although it cannot be reclaimed by the unification through the pointer. Thus the data, which is an atomic or a pointer, should have MRB indicating whether it is pointed by multiple

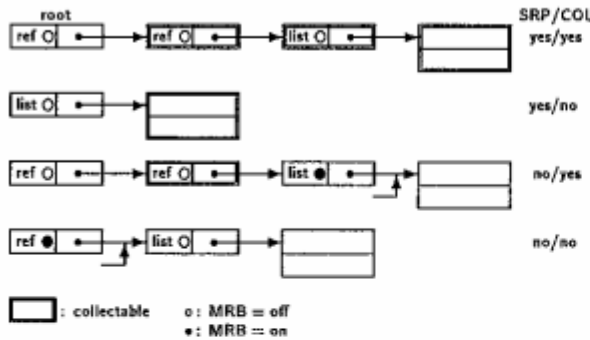


Figure 7: Pipelined Dereference Supporting Incremental Garbage Collection

pointers as shown in Figure 6(e) and (f).

The incremental garbage collection is mainly performed when the unifier makes dereference. A member of the chain of reference pointers can be reclaimed if both its MRB and that of its predecessor are off. The terminal of the chain is also reclaimable if the same condition is satisfied. Especially, all of the members on the chain is collectable if their MRBs are off.

In order to support the reclamation, the pipelined dereference mechanism of PU maintains the following information (Figure 7).

SRP (Single Reference Path):

MRBs of all the pointers on the chain are off.

COL (Collectable):

MRBs of the first two pointers are off.

These are not only passed to the E stage, but also combined with the data typing result to make microprogram entry address, in order that the E stage easily decide whether the reclamation can be done.

On passive unification, if the dereference result is a structure, the structure will be collected after the commit operation. For example, the instruction "collect_list" is located at the beginning of the body code for a clause having head unification with a list cell, and reclaims the list cell if the path to it is single. For the processes filtering streams represented by lists, "reuse_list" is used for passing the list cell directly rather than putting and getting it to/from the free cell pool. To these instructions, SRP is passed through the MRBs of their operands, A_n/X_n .

SRP is also examined by built-in predicates for optimization [Inamura *et al.* 1989]. For example, "set_vector_element" updates an element of a vector to make its new version, providing the path to the vector is single. The stream merger also examines the state of the paths to the variable cell representing a stream and the list cell to be put, in order to reuse these cells.

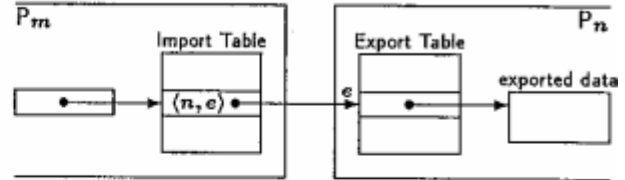


Figure 8: Export Table and Import Table

3.2 Remote Incremental Garbage Collection

As MRB overflows when an object has two or more pointers, there are garbages which cannot be reclaimed by the incremental garbage collection mechanism described in 3.1. Therefore, a PE may exhaust its local memory space and invoke a batch-mode garbage collector. In order to allow the garbage collector to move data around in the local memory space, remote references are indirected with *export table* as shown in Figure 8 [Ichiyoshi *et al.* 1987]. A remote reference consists of the pair of identifiers for PE and the export table entry from which the exported data is pointed. Thus, a PE is free to perform batch-mode garbage collection independently, because other PEs are ignorant of local data addresses but aware of positions of the table entries which never move.

The other indirection table for remote references, *import table* in Figure 8, is introduced to reclaim export table entries incrementally. Entries for single-referenced objects are easily reclaimed using MRB scheme. When a PE, P_e , exports the pointer to a single-referenced object to another PE, P_i , it registers the pointer into *MRB-off* export table. P_i also registers the remote reference into *MRB-off* import table in order to identify that the remote path is single. Unless P_i duplicates the path to the import table entry, the export table entry is reclaimed when P_i makes a remote access to the object. For example, when P_i wants to read the object, it sends a message to get the object. The message also says that the remote path is single, and causes reclamation of the export table entry by P_e . On the other hand, if P_i makes multiple paths and then loses them all, the reclamation is triggered by batch-mode garbage collector on P_i . After the *marking* of the garbage collection, the import table is scanned to find out unmarked entries and send a message for each of these entries for the reclamation of corresponding export table entry.

In order to reclaim export table entries for multiple-referenced objects, we introduced *Weighted Export Counting (WEC)* method [Ichiyoshi *et al.* 1988], which is also independently proposed in [Watson and Watson 1987]. A PE pair, P_e and P_i , exporting and importing the pointer to a multiple-referenced object has entries

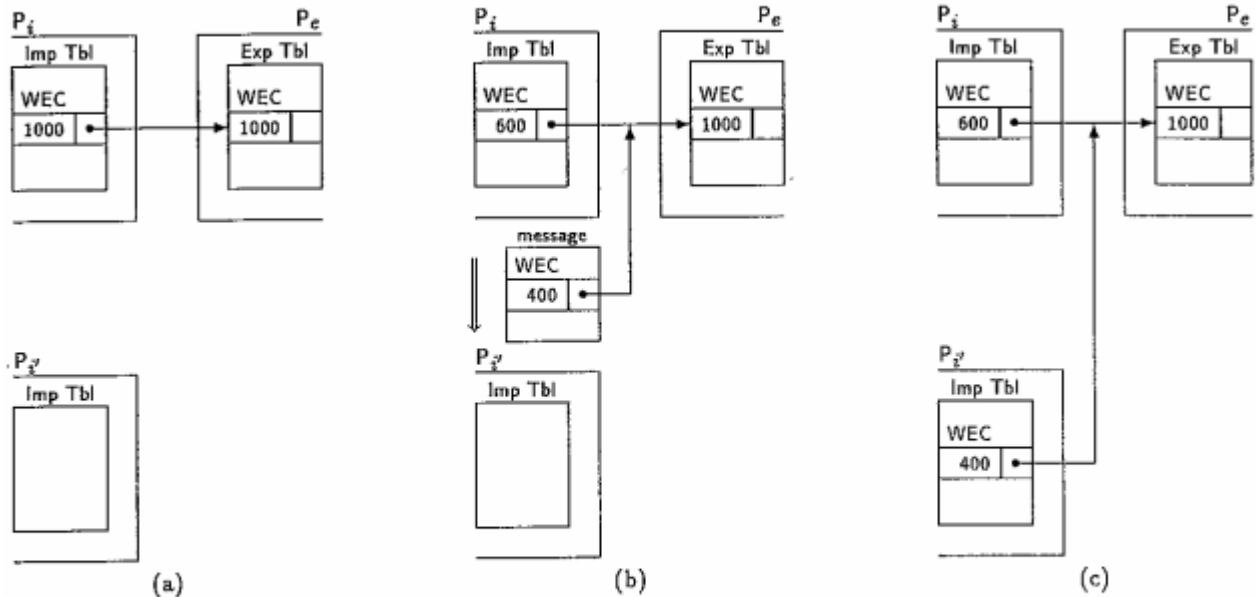


Figure 9: Weighted Export Counting (WEC)

on *MRB-on* export/import table for the object. Each entry has a slot for WEC value, which is a kind of reference counter but is initiated with some large number, say 1000, rather than one, as shown in Figure 9(a). When P_i duplicates the pointer and exports one of the results to another PE, P_e , it divides the WEC value into two portions, say 600 and 400. Then P_i sends a message containing the remote reference and the WEC value 400 to P_e (Figure 9(b)). P_e receives the message and makes an import table entry with the WEC value 400 (Figure 9(c)). Note that the total of WEC values associated with the remote references is equal to the WEC value of export table entry through all phases shown in Figure 9. If P_i (or P_e) finds that there are no paths to an import table entry on incremental or batch-mode garbage collection, it sends a message for reclamation to P_e with the WEC value. The WEC value of the export table entry is decremented by that in the message, and the entry is reclaimed if the WEC value becomes zero.

This scheme has the advantage of ordinary reference counting, because it omits request and acknowledgment messages which the ordinary scheme requires when P_i exports the pointer to P_e . That is, in the ordinary scheme, P_i should send a message for incrementing the reference counter to P_e , and suspend exporting until it receives acknowledgment from P_e . Unless P_i wait for the acknowledgment, the counter on P_e might be cleared transitively by the decrement request from P_e which possibly reaches P_e earlier than the increment request from P_i .

A similar *weighted* counting method, *Weighted Throw Counting* (WTC), is adopted to detect the termination

of a group of goals [Rokusawa *et al.* 1988]. KL1 has the capability to supervise goal groups, called *Shōen*, as if they are meta-interpreted [Chikayama *et al.* 1988]. For example, the operating system PIMOS can detect the termination of a user program represented as a *Shōen*. Since goals in a *Shōen* may be distributed to many PEs, some remote reference counting is necessary to detect the termination of them all. As WEC for remote references, WTC values are given to PEs executing goals in a *Shōen*. Thus, PEs can exchange goals with some WTC values omitting requests/acknowledgments as described before. This feature is very important for efficient execution because an active unification with a remote variables is a goal.

3.3 Multiple Export and Import

Once a multiple-referenced object is exported, it is often exported again. If such an object is repeatedly exported overlooking that it has been already exported, each time an export table entry is consumed. A PE importing such an object repeatedly, worse still, gets multiple copies of the object. In order to solve these problems, both export table and import table are content addressable by hashing. The hash table-for export associates (local) addresses of exported objects with export table entries, while that for import associates remote references with import table entries.

This scheme, however, cannot deal with more complicated situations. For example, if P_i imports two pointers from P_e and P_e' , and each pointer refers to a copy of a

Table 1: Single Processor Performance

benchmark	condition	PIM/m	Multi-PSI/v2	$\frac{\text{Multi-PSI/v2}}{\text{PIM/m}}$
append	1,000 elements	1.63 msec	7.80 msec	4.8
best-path	90,000 nodes	142 sec	213 sec	1.5
pentomino	8 × 5 box	107 sec	240 sec	2.2
15-puzzle	5,885 K nodes	9,283 sec	21,660 sec	2.3

data structure on each PE, P_i will get multiple copies from P_e and P_e' . This troublesome situation may occur in distribution of program codes which have intricate cross references.

Therefore, we introduced global identification of code modules to promise that a PE should not have multiple copies of a code module [Nakajima *et al.* 1989]. When P_e is requested by P_i to send a data object and find out that the object is a code module, it transmits the module identifier rather than the module itself as the reply. Then P_i looks up a hashed table for modules resident in it with the identifier. If the module is resident, P_i simply executes it. Otherwise, P_i sends a special message for getting the module itself to P_e .

4 Performance Evaluation

4.1 Single Processor Performance

Table 1 shows the single processor performance of PIM/m for four benchmarks. The table also includes the performance of Multi-PSI/v2 and the ratio of PIM/m and Multi-PSI/v2 (M/P-speedup) to show the effect of architectural improvement.

The performance for *append* represents the *peak* performance which is 4.8 times as high as that of Multi-PSI/v2. This improvement should greatly owe to pipelined data typing and dereference, because the speedup factor for major E stage operations is only 1.5 (two 65 ns cycle versus one 200 ns cycle). The effectiveness of pipelined dereference supporting the incremental garbage collection is proved by the fact that the speedup factor is significantly larger than 4.2* for Prolog *append* on PSI-II and PSI/UX whose CPU kernels are those for Multi-PSI/v2 and PIM/m respectively [Nakashima *et al.* 1990].

On the other hand, the absolute performance, 615 KLIPS, is still lower than 1.4 MLIPS for Prolog on PSI/UX. A part of this dereference is caused by the fact that the incremental garbage collection mechanism inherently requires additional memory accesses to free

cell pool and variables excluded from list cells. In fact, KL1 *append* performs 10 memory accesses per one reduction in our system, while Prolog *append* does 6 accesses required essentially. The other part, however, should be due to the hardware support for the incremental garbage collection which is not yet sufficient to remove the overhead. For example, we estimated that some modifications of the hardware with few gates for;

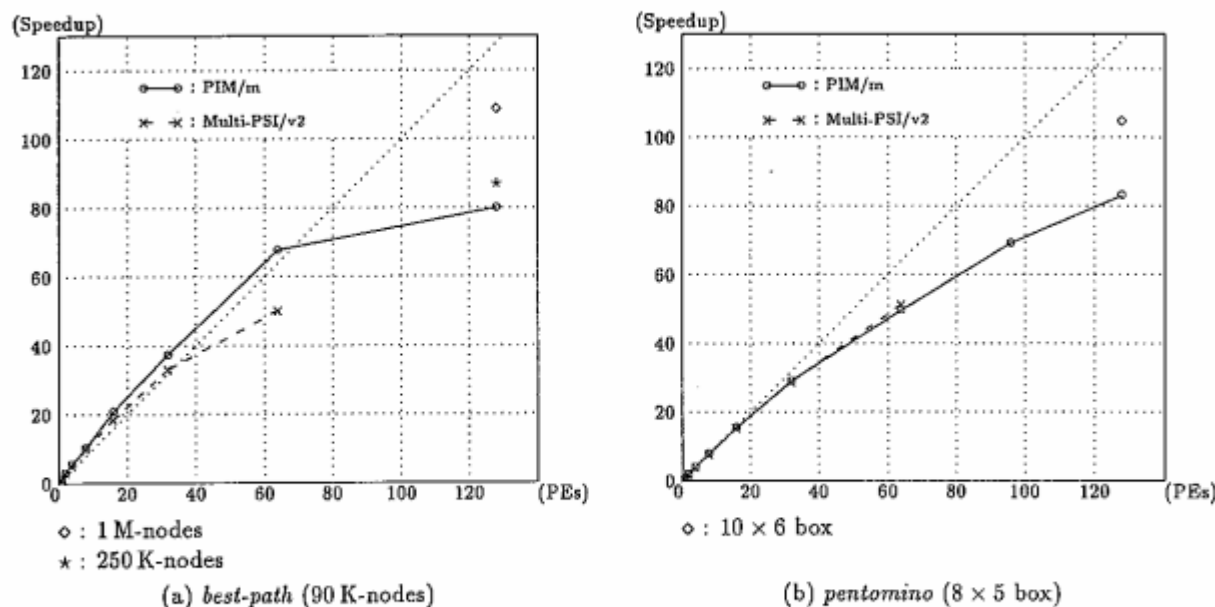
- making information indicating whether the dereferenced path is totally collectable,
- fetching an element from free variable pool in pipeline, and
- storing the result of an ALU operation into both the structure pointer and an argument register

will make the performance 810 KLIPS.

The other three benchmarks are search programs with various parallel algorithms and load distribution strategies. *Best-path* finds out the shortest path between two vertices of a directed weighted graph with a parallelized Dijkstra's algorithm and static load distribution [Wada-K and Ichiyoshi 1989]. *Pentomino* makes OR-parallel exhaustive search to solve a packing piece puzzle problem with a multiple level dynamic load distribution method [Furuichi *et al.* 1990]. *15-puzzle* solves a well-known puzzle problem in parallel by employing iterative-deepening A* algorithm [Wada-M and Ichiyoshi 1991]. Although these programs are not practical, the algorithms and load distribution strategies should be generally adopted to various application programs of parallel processing. Thus, it is expected that the performance for them reflects the performance sustainably gotten on PIM/m.

The M/P-speedup for these program, 1.5 to 2.3, are not excellent in contrast with the case of *append*. This is probably caused by two major reasons, context switch and cache miss. In these programs, context switches frequently occur, every two to three reductions, by the termination or suspension of goals, while never in *append*. Since instructions for the context switch take dozens of cycles for execution in the E stage and make pipeline stagnant, the pipelined architecture doesn't gain much performance improvement for these programs.

*This value is normalized to compensate the machine cycle difference between Multi-PSI/v2 and PSI-II.

Figure 10: Speedups for *best-path* and *pentomino*

Cache miss penalty should be the major degradation factor in *best-path* which has a large working set. Even in Multi-PSI/v2, cache miss degrades the performance 10 to 20% as reported in [Nakajima and Ichiyoshi 1990]. Thus, the penalty relative to the machine cycle becomes more critical, because the cache size and physical memory access time of PIM/m are not greatly evolved from Multi-PSI/v2.

4.2 System Performance

System performance is strongly related with load distribution strategy and communication cost. Since PIM/m has four times as many PEs as Multi-PSI/v2 has, it might become difficult to balance loads distributed to PEs. As for communication cost, we evaluated that the network capacity of Multi-PSI/v2 is much larger than required [Nakajima and Ichiyoshi 1990]. Therefore, we designed PIM/m's network making its throughput and bandwidth almost equal to those of Multi-PSI/v2's, expecting that the network still has enough capacity. The frequency of message passing, however, might be contrary to our expectation, because of underestimation of hot spot effect and so on.

The speedup, which is gotten by dividing execution time for single processor by that for n processors, may give preliminary answers about those questions. Figure 10 shows the speedups of PIM/m and Multi-PSI/v2 for *best-path* and *pentomino*. Up to the 64 PE system,

the speedup of PIM/m are quite similar to or slightly better than that of Multi-PSI/v2. Especially, the result of *best-path* shows surprising super-linear speedup, probably because partitioning the problem makes required memory space for a PE small and reduces cache miss rate and/or the frequency of batch-mode garbage collection. These results show that the network of PIM/m stands increase of message passing frequency caused by the improvement of PE performance. Thus, the performance of single cabinet minimum system is greatly improved from Multi-PSI/v2. That is, M/P-speedup is 5.6 for *best-path* and is 8.3 for *pentomino*.

On the other hand, the speedup of the 128 PE system are considerably low, especially for *best-path*. Thus, the M/P-speedups for 4-cabinet a half of maximum system are 3.7 for *best-path* and 6.4 for *pentomino*. This implies that the problem size is too small to distribute loads to 128 PEs and/or the message passing frequency exceeds the network capacity. As for *best-path*, the reason of low speedup seems to be small size of the problem which takes only 1.8 sec on the 128 PE system, because a PE transmits messages only to its adjacent PEs. For example, when the problem is scaled up by increasing the number of nodes from 90 K to 250 K and 1 M, the speedups for the 128 PE system become 87 and 109 respectively, as shown in the figure*.

*Since large problems cannot run on small size systems, the speedups are estimated by multiplying 32 PE speedups for small problems by 32 to 128 PE speedups for large problems.

In *pentomino*, its load distribution strategy might cause hot spot PEs which pool loads and distribute them in demand driven manner. The hot spot, however, is possibly that of computation for load generation rather than communication for distribution. The problem size may also limit the speedup, because the execution time of the 128 PE system is only 1.3 sec. The speedup of larger size problem, which is for 10×6 box and takes 211 sec on the 128 PE system, is 105 as shown in the figure*. We are now planning further evaluation and analysis to confirm these observations or find out other reasons.

As for *15-puzzle*, we measured the speedups of 64 and 128 PE systems changing the problem size as shown in Figure 11. The figure also shows the number of nodes in the search space for each of seven initial states of the game board. The results for the 64 PE system of PIM/m is also quite similar to that of Multi-PSI/v2. The speedup of the 128 PE system, 38.7 to 109.2, are tightly related to the size of problems. The analysis of this relation is also left as a future work.

5 Concluding Remarks

This paper presented the hardware architecture of PIM/m system, its processor element, and the pipelined micro-processor dedicated to the fast execution of KL1 programs. The KL1 implementation issues focused on its relation with garbage collection were also described. Then preliminary performance evaluation results were shown with brief discussions on them.

We are now planning a research concentrated on further evaluation of the performance of PIM/m and the behavior of various KL1 programs. The evaluation results and detailed analysis on them should greatly contribute not only to the performance tune-up of PIM/m but also to the research on parallel inference machines in next step.

Acknowledgment

We would like to thank all those who contributed to the development of PIM/m system in ICOT, Mitsubishi Electric Corp. and related companies. We also wish to thank Vu Phan and Jose Uemura for their contribution to this paper.

References

- [Chikayama 1984] T. Chikayama. Unique Features of ESP. In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 292-298, Nov. 1984.
- [Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. 4th Intl. Conf. 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 *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, 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 *Proc. 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 50-59, Mar. 1990.
- [Ichiyoshi et al. 1987] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proc. 4th Intl. Conf. on Logic Programming*, pp. 257-275, 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. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 904-913, Nov. 1988.
- [ICOT 1990] ICOT. *Proc. Workshop on Concurrent Programming and Parallel Processing*, 1990.
- [Inamura et al. 1989] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization Technique Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proc. North American Conf. on Logic Programming 1989*, pp. 907-921, 1989.
- [Kimura and Chikayama 1987] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proc. 4th IEEE Symp. on Logic Programming*, pp. 468-477, Sept. 1987.
- [Machida et al. 1991] H. Machida, H. Andou, C. Ikenaga, H. Nakashima, A. Maeda, and M. Nakaya. A 1.5 MLIPS 40

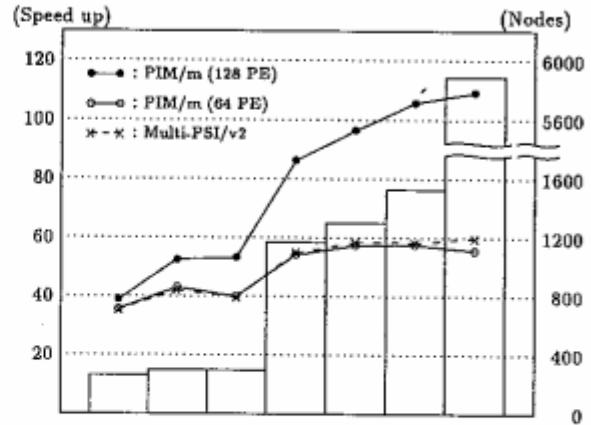


Figure 11: Speedup for 15-puzzle

- bit AI Processor. In *Proc. Custom Integrated Circuits Conf.*, pp. 15.3.1–15.3.4, May 1991.
- [Masuda *et al.* 1988] K. Masuda, H. Ishizuka, H. Iwayama, K. Taki, and E. Sugino. Preliminary Evaluation of the Connection Network for the Multi-PSI system. In *Proc. 8th European Conf. on Artificial Intelligence*, pp. 18–23, 1988.
- [Nakajima *et al.* 1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. 6th Intl. Conf. and Symp. on Logic Programming*, 1989.
- [Nakajima and Ichiyoshi 1990] K. Nakajima and N. Ichiyoshi. Evaluation of Inter-Processor Communication in the KL1 Implementation on the Multi-PSI. In *Proc. 1990 Intl. Conf. on Parallel Processing*, Vol. 1, pp. 613–614, Aug. 1990.
- [Nakashima and Nakajima 1987] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *Proc. 4th IEEE Symp. on Logic Programming*, pp. 104–113, Sept. 1987.
- [Nakashima *et al.* 1990] H. Nakashima, Y. Takeda, K. Nakajima, H. Andou, and K. Furutani. A Pipelined Microprocessor for Logic Programming Languages. In *Proc. 1990 Intl. Conf. on Computer Design*, pp. 355–359, Sept. 1990.
- [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 *Proc. 1990 Intl. Conf. on Parallel Processing*, Vol. 1, pp. 18–22, Aug. 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. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 978–986, Sept. 1988.
- [Taki *et al.* 1984] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima, and A. Mitsuishi. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 398–409, Nov. 1984.
- [Taki 1988] K. Taki. The Parallel Software Research and Development Tool: Multi-PSI System. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*. North-Holland, 1988.
- [Uchida *et al.* 1988] S. Uchida, K. Taki, K. Nakajima, A. Goto, and T. Chikayama. Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project. In *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 16–36, Nov. 1988.
- [Ueda 1985] K. Ueda. Guarded Horn Clauses. Technical Report 103, ICOT, 1985. (Also in *Concurrent Prolog: Collected Papers*, The MIT Press, 1987).
- [Wada-K and Ichiyoshi 1989] K. Wada and N. Ichiyoshi. A Study of Mapping of Locally Message Exchanging Algorithms on a Loosely-Coupled Multiprocessor. Technical Report 587, ICOT, 1989.
- [Wada-M and Ichiyoshi 1991] M. Wada and N. Ichiyoshi. A Parallel Iterative-Deepening A* and its Evaluation. In *Proc. KLI Programming Workshop '91*, pp. 68–74, May 1991. (In Japanese).
- [Warren 1983] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, Oct. 1983.
- [Watson and Watson 1987] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architecture. In *Proc. Parallel Architecture and Languages Europe*, June 1987.