TR-486

The Multi-PE Data Processing and Its Evaluation

by
K. Nakajima

July, 1989

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

The Multi-PSI : Inter-PE Data Processing and Its Evaluation

Katsuto Nakajima
Institute for New Generation Computer Technology

Abstract

The Multi-PSI system developed in the Japanese FGCS project is a loosely coupled multiprocessor and is used as a testbed to experiment on the implementation of parallel execution of KL1 programs.

The key point of the implementation is to minimize the inter-processor communication because the cost for data access between processors is much higher than inside one processor.

Address translation tables for inter-processor data references, called *export* and *import tables*, are introduced to realize local garbage collection. The *weighted reference counting (WEC)* method is used for incremental inter-processor garbage collection. To avoid redundant copies of KL1 data among processors, the two schemes, *export/import hash tables* and *structure ID*, are employed.

The cost of handling frequent messages is 25 to 80 μsec, and they are expected to be reduced to around 70 to 85 % of the current implementation by tuning up. Evaluation with two medium size benchmark programs shows the overhead for inter-processor communication is about 10 to 25 % in operating time even with the large cost of communication processing. The network traffic is very light and the topology is not limiting the system performance in the current system.

1 Introduction

In the course of the Japanese Fifth Generation Computer Project, a prototype parallel inference system, the Multi-PSI[Taki 88], was developed aiming at two objectives: (1) to provide a practical tool for parallel software research and development, (2) to provide a testbed for implementation of parallel execution of KL1 programs. KL1 is a stream AND-parallel logic programming language based on Flat GHC[Ueda 86].

The Multi-PSI is a non-shared memory multiprocessor, whose processing elements (PEs) are the CPUs of the personal sequential inference (PSI) machine. There was an earlier version, the Multi-PSI/V1, but in this paper the name Multi-PSI represents only the Multi-PSI/V2.

Up to 64 PEs are connected to each other to form a two-dimensional mesh network that can switch messages and perform automatic routing. The two-dimensional mesh may not be the best network topology for parallel execution of KL1, but it is a realistic candidate for dense implementation in a limited space retaining the scalability with the currently available technology. Therefore, the Multi-PSI can be considered a prototype of a highly parallel machine, in which the number of processing nodes is, for instance,

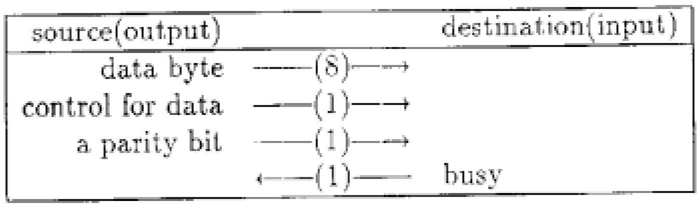

Figure 1: Physical Lines of the Network Channel

more than 1000. In the design of KL1 implementation, the emphasis was placed on achieving efficient inter-PE communication and efficient use of local memory.

This paper describes the design issues of the KL1 processor especially on the inter-PE data processing, and also gives some evaluation results on the inter-PE processing and the dynamics of parallel execution.

2 Overview of the Multi-PSI Architecture

2.1 Processing Element Architecture

The processing element (PE) of the Multi-PSI is the CPU of the PSI-II [Nakashima 87]. It is a horizontal micro-programmed CISC architecture which enables a flexible implementation suited for incrementally enhancing the performance and adding various functions. The cycle time is 200 nsec. It has a 4-Kword direct map cache memory.

2.2 Network Controller Architecture

Each PE is associated with a specially designed network controller. It has five pairs of channels connected to the four adjacent network nodes and to the PE of the node. A pair of channels is for input and output, each of which has 9-bit data with a parity line and one bit busy acknowledge signal (Figure 1). A 48-byte buffer (Output Buffer) is installed at each output channel to retain messages in the event of the destination node being busy. A 4-Kbyte buffer (Write Buffer and Read Buffer) is installed at each input and output channel for the PE of the node to reduce disturbance of processing.

The controller also has an automatic routing function. It provides two transmission modes. One is to route messages according to the *physical PE number* in the message header. The other uses the *logical PE number*, a point in a hypothetical two-dimensional space, named the processing power plane (P^3), for automatic load balancing[1][Takeda 88]. A software-defined table called a *path table* is looked up to determine transmission direction. The bandwidth of each channel is 5 Mbytes/sec.

2.3 Routing Strategy

A fixed routing strategy called *prioritized coordinate ordering* is adopted. It means transmitting along the *x-coordinate* until the distance in the coordinate becomes zero, then transmitting along the *y-coordinate*.

[1]The P^3 scheme is currently being studied and has not been implemented yet.

This prevents network deadlock as it is impossible to make a circle of nodes waiting for one another. Note that even with this strategy, a deadlock may occur if a PE fails to take in all the messages directed to it.

2.4 Basic Message Handling Routine

A low level microcoded routine is responsible for basic handling of the messages to and from the network controller. It performs composition and decomposition of message packets such as handling message header and tail, and arranging a 32-bit data in the processor to/from four byte serial data in the network controller. The operations in this routine are independent from the implementation of KL1.

The following is also a function of this routine. When one complete message (from the head to the tail) arrives at the Read Buffer, an interrupt is signaled to invoke the routine, which moves the contents of the Read Buffer to a large pre-defined memory area, called Read Packed Buffer. The aim is to prevent network deadlock by propagating the processing delay to other nodes along the network path.

On sending a message, if the routine cannot find enough room to store the whole message in the Write Buffer, it will wait until more room becomes available.

3 KL1 Distributed Implementation

3.1 Key Points of the Design

The architecture of the next-generation parallel inference system, PIM[Goto 88], has two levels; the upper level is a scalable network-connected multi-cluster architecture, and the lower level (cluster) is shared memory high-performance architecture. As a PE in the Multi-PSI corresponds to a cluster in the PIM, the research of the language implementation on the Multi-PSI has been concentrated on the problems related to inter-cluster non-shared memory processing, and has employed various mechanisms to solve them.

The mechanisms we designed are meant to be available for a highly parallel system with more than one thousand nodes, and not necessarily optimized for the Multi-PSI itself which has up to 64 nodes.

The most critical issue in the design is that the cost of data access between PEs is much higher than that within one PE. So our first priority was to reduce inter-PE communication.

Inter-PE communication can be categorized into two: inter-PE data management and inter-PE goal management (control of the distributed computation). This paper will not describe the latter, which can be found in [Ichiyoshi 87], [Rokusawa 88], and [Nakajima 89].

The key points in our design for inter-PE data management are as follows:

(1) Local garbage collection (local GC)

(2) Efficient inter-PE garbage collection (inter-PE GC)

(3) Avoidance of redundant data copies

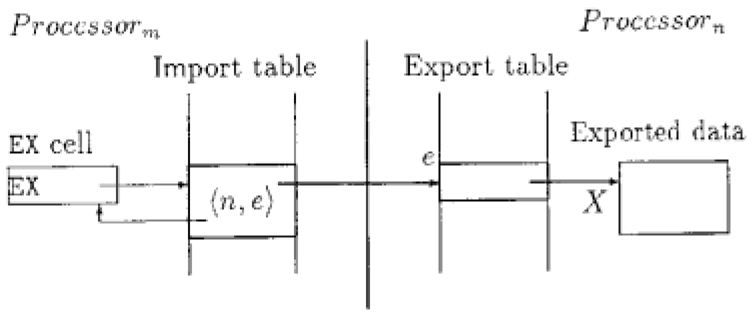

Figure 2: External Reference and Export/Import Table

3.2 Inter-PE Data Reference

KL1 goals can be moved to other PEs by `throw_goal` messages initiated by KL1 pragma $(\ldots, goal@processor(PE), \ldots)$[Ichiyoshi 87, Nakajima 89]. When a KL1 goal containing pointers to a data structure (including unbound variable) is moved to another PE, such pointers become a pointer referring to a data structure in a different PE. This is called an *external reference*, and the pointer is called an *external (reference) pointer*. To generate an external pointer is called *exporting*, and to receive the external pointer is called *importing*.

As the single assignment semantics of KL1 allows data copying, 1-word atomic values such as integers and atoms are usually copied and carried with the thrown goals. However, if the data is a structure, the overhead for transferring the whole data may become unnecessarily large; the goal might not need all the data. Therefore, the system exports structure data as an external pointer and the contents are transferred lazily on request by a `read` message. To retain the identity of variables, unbound variables are always exported as external pointers.

3.3 Local Garbage Collection

To collect all the garbage in the system, a *global GC* is needed, which must be performed by cooperation of all the PEs. This global GC has the serious disadvantage of forcing all the processors to stop program execution. It takes a very long time, up to the longest path of data references in the system multiplied by the network communication delay. The more processing nodes the system has, the more serious the disadvantage will be. It is better to perform GC locally as far as it can reclaim enough memory resource. To collect garbage without communicating with other PEs is called *local GC*.

To perform local GC, all the exported data must be known by the PE so that they are not thrown away. For this purpose, the *export table* keeps addresses of all the exported data in a PE. All the external pointers point to the entries of the table from outside the PE(Figure 2). The external pointers are represented in the form $< n, e >$, where n is the exporting PE number and e is the entry position in the export table.

The entries of the export table are maintained on local GC according to the movement of the exported data. The maintenance cost of the export table on each export or on local GC and the indirection cost of the external accesses are the overhead of this scheme, but would be well compensated for by the avoidance of performance degradation caused by global GC.

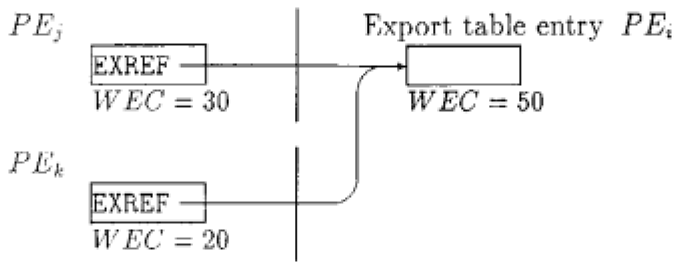

Figure 3: Weighted Export Counting Scheme

3.4 Efficient Inter-PE Garbage Collection

To reclaim garbage cells pointed to by the export table, the entries of the table must be deleted when they become garbage. Three typical schemes can be considered. One is the stop GC, which includes the global GC as described above, and may be the worst in terms of system performance. Another is on-the-fly GC, in which dedicated processors are running for memory reclamation. This scheme is often favorable for ensuring real-time response, but the total performance is much reduced and the mechanism is hard to implement. The last is incremental GC, probably using *reference counting*, which is good in memory access locality, but not in terms of run-time overhead.

The major defect of reference counting is that both when an external pointer becomes garbage (decrease of references), and when an external pointer is copied (increase of references), a message must be sent from the importing PEs to the exporting one to indicate the change of the references. Furthermore, an acknowledgment message is necessary to prevent the reference count to reach zero by accident, because increment and decrement messages may arrive the exporting PE under certain racing condition.

To solve these problems, we employed the weighted export counting (WEC) method [Ichiyoshi 88] based on the weighted reference counting (WRC) principle [Watson 87] for incremental inter-PE GC. In this scheme, a weighted count of a positive integer, called `WEC`, is kept on the both export and import sides (Figure 3). The system operates keeping the following invariant:

$$\begin{aligned}\texttt{WEC of X at exporting PE} = {} & \textstyle\sum_{at\ importing\ PEs} (\texttt{WEC of X}) \\ & + \textstyle\sum_{in\ network} (\texttt{WEC of X})\end{aligned}$$

The WEC scheme has the following advantages:

- When an imported pointer is copied to another PE, the `WEC` is split. No message is sent to maintain the `WEC` of the exporting PE.
- There is no racing problem.

To keep `WEC` on the importing side, we have the *import table*(Figure 2). The import table is also beneficial in accelerating inter-PE GC after a local GC. It is swept after a local GC, and `release_exref` messages are sent to the exporting PEs to return the `WEC`s if the imported pointers are known to be no longer used.

3.5 Re-exporting

The same data structure may be exported to the same PE several times. If a different external pointer is given to the re-exported data, the importing PE cannot recognize it as a pointer to the same data and may send a `read` message many times, in which case the data is transferred redundantly. This can be avoided by reusing the same export and import table entries. For this purpose, the *export hash table* and *import hash table* are provided on each side. The export hash table associates the exported data addresses with their external pointers, and the import hash table associates the imported external pointers with their import table entries. The number of imports through the same import table entry is also held as *import count* in the entry to enable efficient release of the entry.

3.6 Global Structure Management

In our export system, the external pointer is originated at the exporting PE. For example, if PE_B has a structure copied from PE_A, and PE_C has two external references to both the original structure in PE_A and the copy in PE_B, their external pointers are not the same. PE_C will have two copies after reading both of them. Furthermore, when the same data is imported twice even from the same PE, there are no means to realize the re-import if a copy has been created before the second import, because the detection of re-importing is done only for the externally referencing data not copied yet.

If the copied structure is large and will live long, this will cause serious inefficiency in both memory space and data transfer. In the worst case, copies are created on each import if a pair of mutually linked structures are read alternately along a loop. This is not a rare case for program code. To solve this problem, we introduced *structure ID* for such structures, which is a global ID attached to exported structures. By this means, what was originally the same structure is duplicated only once in a PE even if it is imported more than once from different PEs.

3.7 White and Black Exports

The external reference management described so far maintains both the `WEC` and import count, and looks up the hash table on each export and import.

Fortunately, the MRB mechanism[Chikayama 87] can be used to optimize this. To export a single reference pointer at low cost, a simplified pair of export and import tables, called *white export* and *white import tables*, are used. The original tables are called the *black export* and *black import tables*. From our observation, pointers copied once will often be copied again later. In contrast, a single reference pointer is not likely to be duplicated after being exported. Thus, the white export and import tables do not have hash tables because the exported pointers are rarely exported again.

The white import table can be considered as an import table for the pointers whose WEC and import count equals one, and its entries are released immediately when the imported pointers are collected by MRB GC[2]. White export entries are also released only when the `release_exref` message is received.

[2]If an imported pointer is copied, the MRB of both the original and copied pointers is turned on so that the import table entry will not be released when one of the pointers becomes garbage.

4 Evaluation

This section shows the costs of basic inter-PE processing overhead and gives some measurement results of dynamic characteristics through execution of two experimental programs.

4.1 Communication Overhead

Figure 4 shows the micro instruction steps (or time in microseconds) of the costs for handling typical messages. The data was obtained by microprogram tracing and does not include the memory access overhead (cache miss penalty). The external pointers in this evaluation are exported through the white export table.

The costs of sending and receiving a `throw_goal` message whose arguments are an atom and two external reference pointers in a typical situation[3] are shown in Figure 4(a) and (b). It takes about 85μsec for sending such a goal and about 130μsec for receiving and storing it for later execution.

`Copy_to_RPKB` is the cost of moving a 65-byte message from the Read Buffer to the Read Packet Buffer. It can be omitted by decoding messages directly from the Read Buffer when there is enough room in the Read Buffer to get further messages. With this optimization, the cost can be reduced by about 13.5%. Furthermore, according to our recent estimation, the cost of sending it would be reduced by up to 15 % and the cost of receiving it by up to 30 % by fully tuning up the microcode.

Figure 4(c),(d),(e), and (f) describe the cost of sending and receiving of a `read` message requesting the contents of an external pointer and an `answer_value` message answering the request. The returned data is a list whose CAR and CDR are an atomic data and another external pointer respectively.

It costs 25μsec to send a 14-byte `read` message and 35μsec to receive it. It costs 42μsec to send a 24-byte `answer_value` message and 80μsec to receive it. In these cases, our estimations of the tuning up effect are also 15 to 30 % of the current version.

In the other evaluation, the overhead of black export/import is known to be 2 to 5.6 μsec (10 to 28 steps) more than that of white export/import in `read` or `answer_value` message handling.

4.2 Dynamic Characteristics of Inter-PE Communication

4.2.1 Benchmarks

The two medium size programs listed below were used to analyze the dynamic characteristics of inter-PE communication.

- **Pentomino :** A packing pieces puzzle program which packs pieces of various shapes into a given rectangle.
- **Bestpath :** A program to find the lowest cost path from a given node to all other nodes of a network consisting of 330 by 330 mesh-connected nodes.

[3]The PE receiving the message is assumed to already have program code and management information to execute the goal.

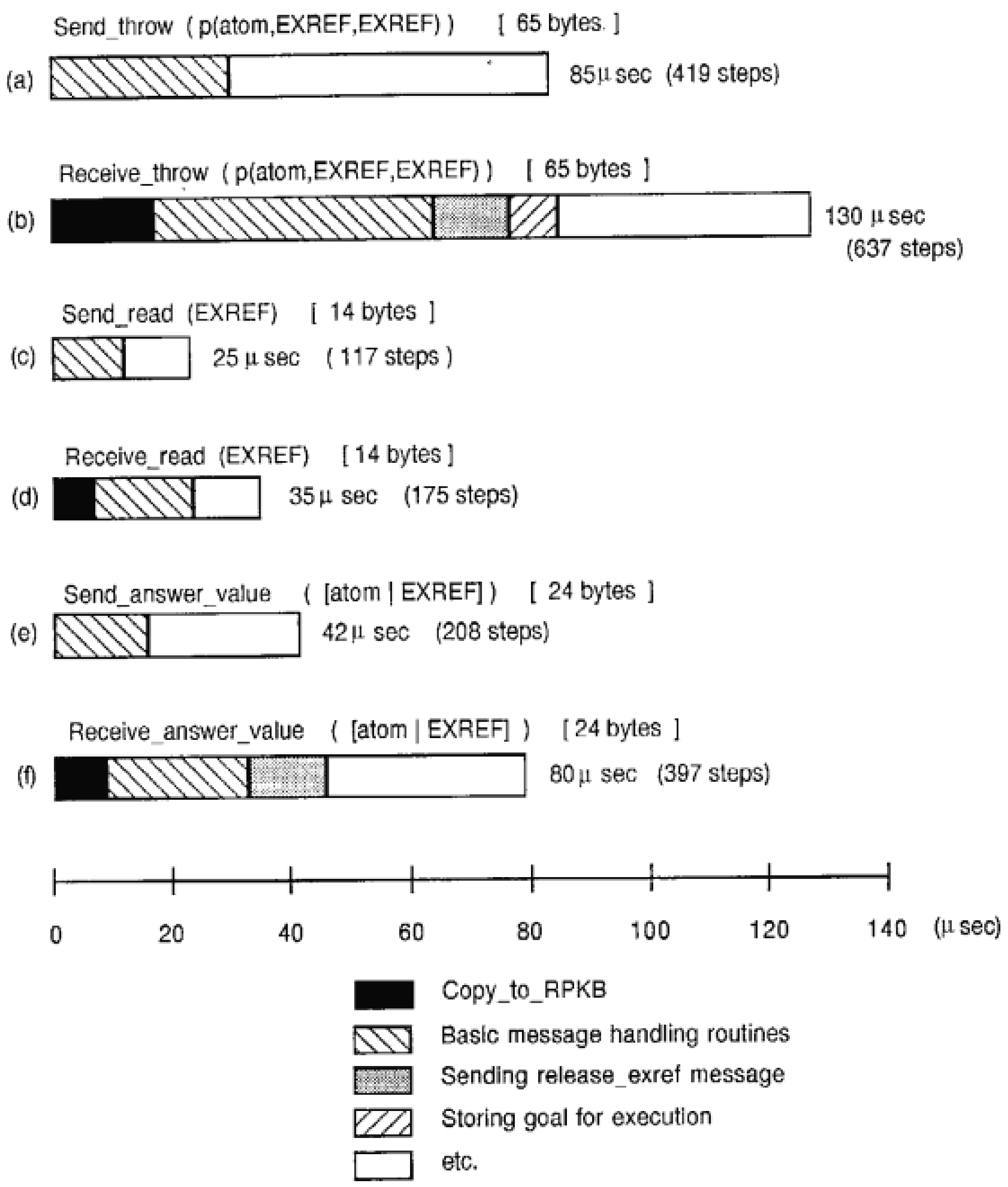

Figure 4: Message Handling Cost

Pentomino on 16 PEs		
read	183,405	(34.0 %)
answer_value	183,399	(34.0 %)
release_exref	165,474	(30.7 %)
unify	2,171	(0.4 %)
throw_goal	2,171	(0.4 %)
(others)	2,765	(0.5 %)
Total messages	539,385	
Total reductions	14,735,231	
Net execution time	31,482	(msec) = 468 KRPS
Time/reductions	2.14	(μsec)
Reductions/messages	27.3	
Messages/sec	17,133	

Bestpath on 16 PEs		
read	63,385	(28.1 %)
answer_value	53,476	(23.7 %)
unify	41,600	(18.4 %)
release_exref	31,688	(14.0 %)
throw_goal	31,688	(14.0 %)
(others)	3,863	(1.7 %)
Total messages	225,700	
Total reductions	4,900,791	
Net execution time	16,426	(msec) = 298 KRPS
Time/reductions	3.35	(μsec)
Reduction/messages	21.7	
Messages/sec	13,740	

Table 1: Message Frequency and Reductions

They use KL1 pragma to distribute goals over PEs, trying to localize the process communication while keeping enough parallelism.

The Pentomino program does an exhaustive search of an OR tree of possible piece placements. One master PE starts from the root and searches the tree down to a certain fixed depth, and evenly distributes the subtrees below that depth to all PEs including itself.

The execution of the Bestpath problem is divided into two different stages. The first is distribution of the processes that represent the network nodes. The second is communication between such nodes to solve the problem; the minimum cost information known so far is propagated to the adjacent nodes. Only the measurement results of the latter stages are reported here.

A system with 16 PEs was used for the measurement.

4.2.2 Message Frequency and Network Traffic

Table 1 shows the total number of messages in the system. The total number of reductions[4] is also listed.

In Pentomino, the `read`, `answer_value` and `release_exref` messages are dominant. In Bestpath, the `unify` and `throw_goal` messages join them. The average frequency of the messages is 13 to 17 K times/sec. Assuming that the average number of the message length is 20 bytes and the number of message traveling distance (the number of hops of the network channels) is 2.5,[5] the average traffic of a channel is $(20 \times 2.5 \times 17\mathrm{K})/(\texttt{number of channels})^6 = 17.7$(Kbytes/sec), which is less than 0.35 % of the channel bandwidth of 5Mbytes/sec.

The total message frequency in the system is proportional to the number of PEs if the granularity of the distributed processes is the same as in our evaluation. The average number of hops is nearly proportional to the square root of the number of PEs. The number of channels that enlarges the network bandwidth of the system is proportional to the number of PEs. This means that the average channel traffic can grow proportionally to the square root of the number of PEs.

We can improve PE performance by a factor of 8×4 in the PIM system: 8 times by replacing the PE with a shared memory cluster, in which eight processors with coherent cache memories are connected by a common bus; 4 times by using VLSI technology, making each processor four times faster than the PE in the Multi-PSI. Therefore, in a PIM system of 1,600 clusters with a four times faster network, the network traffic would be $(0.35 \times 32 \times \sqrt{100})/4 = 28\%$ of the 20 Mbytes/sec bandwidth, and it would not be problematic at least with these benchmark programs. However, unless the locality of the inter-PE communication is considered or unless hot spots in the network are avoided by program, the network could be saturated quite easily.

4.2.3 Distribution Overhead

Performance of parallel execution on a network-connected multiprocessor like the Multi-PSI can be degraded by the following factors.

(1) Load imbalance: Cost of not being able to keep PEs in operation

(2) Synchronization overhead: Cost of suspending processes because of communication interrupts and cost of resuming them

(3) Communication overhead: Cost of message handling

(4) Network traffic jam: Cost of keeping sender and/or receiver wait

(5) Losing memory locality: Cache miss penalty caused by increased working set by interruption

(6) Speculative computation: Cost of the work that would not have been required if the program had run sequentially.

[4] The number of commitments.

[5] $2 \cdot (L-1) \cdot (L+1)/(3 \cdot L)$ *where* $L = \sqrt{(\texttt{number of PEs})} = 4$. This is assuming that the mesh network is a square and that messages are sent randomly between PEs.

[6] $4 \cdot L \cdot (L-1) = 48$ *when* $L = 4$.

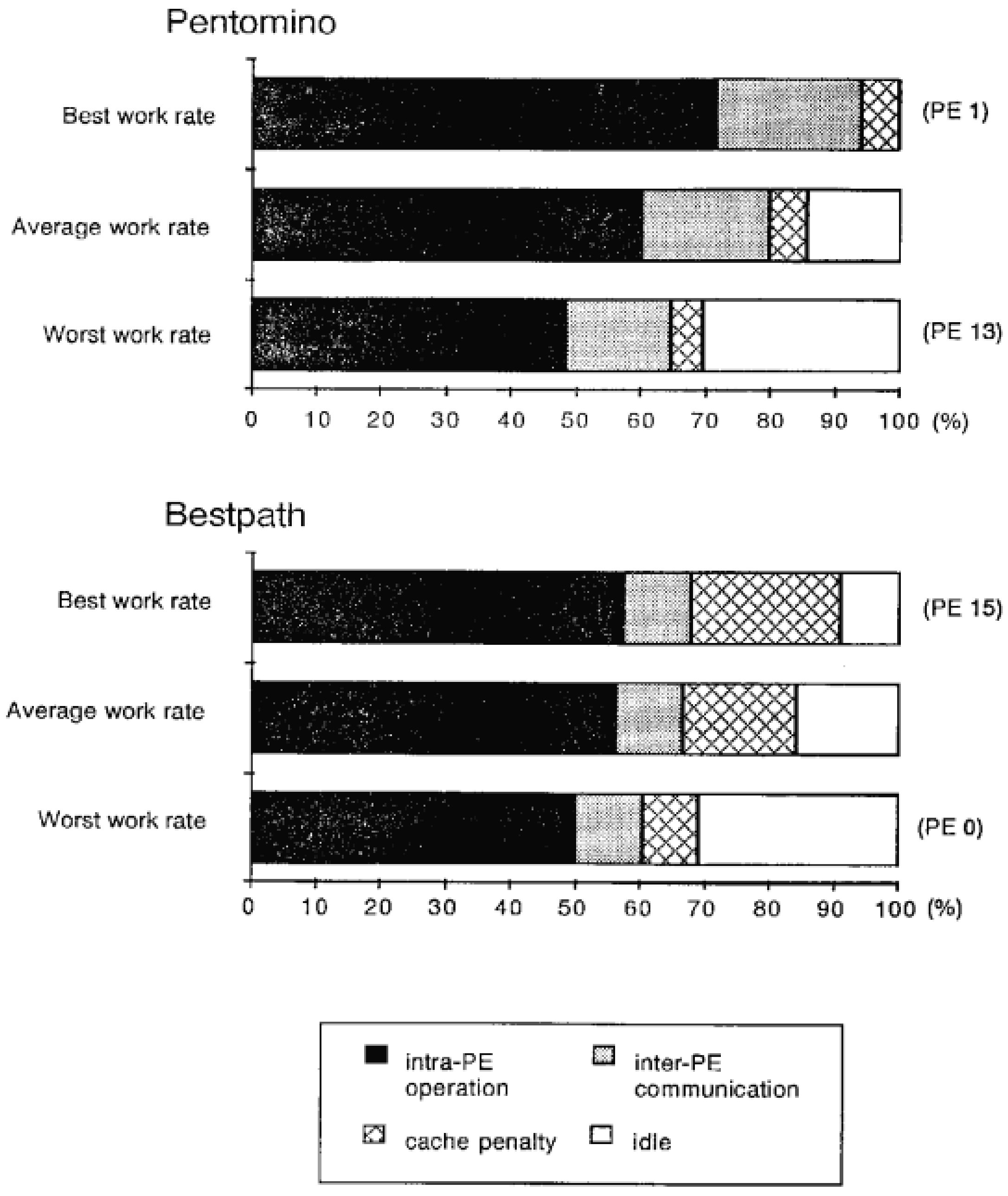

Figure 5: Performance Degradation Analysis

Some of them are related to each other. In our evaluation here, we can omit (4) because the network traffic is quite low. We can probably omit (6) because the two programs are known to do only small amount of speculative computation.

The Multi-PSI has a built-in hardware for counting the number of steps executed at specified micro instructions. We also have calendar clocks in each processors, all of which are simultaneously initialized when the system starts up. By using them and logging the time of entering and exiting from idle status at each processor, we can measure the following items.

(a) Operating steps for effective work (regrettably including the overhead of suspending and resuming processes because the routines for them are common with normal processing)

(b) Operating steps for inter-PE message handling

(c) Total operating time

(d) Total idling time

The total (net) execution time is (c) + (d), and (d) includes idling time because of load imbalance or traffic jam. The *work rate* can be calculated from $(c)/((c)+(d))$. The difference between (c) and (a) + (b) can be considered as the cache miss penalty. (B) is the communication overhead without cache penalty.

Figure 5 shows the result of the system degradation analysis with two benchmarks.

In both benchmarks, the average work rates are about 85 % and the ratios of the intra-PE operating steps are almost the same.

The inter-PE communication steps in Pentomino occupy 25 % of the operating steps, a much larger than 10 % in Bestpath. This is because structure data representing partial solutions are frequently transferred between PEs. In Bestpath, the cache penalty is very large. It tells that the communications between small grain processes representing the network nodes requires a very large working set probably in both intra- and inter-PE processing.

5 Conclusion

This paper described the network architecture of the Multi-PSI and stated the issues of implementing KL1 on a loosely coupled multiprocessor, the Multi-PSI. This paper also revealed the cost of the inter-PE communication and analyzed the sources of performance degradation.

The KL1 distributed system was designed and implemented aiming at minimizing the inter-PE processing cost which may seriously degrade the system performance. The export and import tables are introduced to enable local GC. The WEC is used for incremental inter-PE GC. The export and import hash tables and the structure ID avoid redundant copies of KL1 data.

The cost of one message handling is roughly 10 to 40 times that of an average reduction. The work rate analysis tells us that even with rather large cost of communication, the overhead for inter-PE communication can be kept within one fourth of the operating steps, by considering the locality of the process communication. The network traffic is very low and the topology is not limiting the system performance in the current system with these benchmarks.

At least two programs are proved to run in much scaled-up systems built on the principles of the current system. We believe that this will apply for wide area of application programs.

Acknowledgments

I would like to thank the ICOT Director, Dr. K. Fuchi, and the chief of the fourth research laboratory, Dr. S. Uchida, for giving me the opportunity to conduct this research. I would also like to thank the researchers of ICOT and the cooperating companies, who have worked with me in designing and implementing the KL1 system on the Multi-PSI/V2. Lastly, I appreciate the help of Mr. Onishi and Mr. Imai in analyzing the evaluation data and making figures in this paper.

References

[Chikayama 87] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[Goto 88] 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*, ICOT, Tokyo, 1988.

[Ichiyoshi 87] 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 88] N. Ichiyoshi, K. Rokusawa, K. Nakajima and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Nakajima 89] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.

[Nakashima 87] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine : PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, Sept. 1987.

[Rokusawa 88] 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*, Vol. I, 1988.

[Takeda 88] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Taki 88] K. Taki. The Parallel Software Research and Development Tool: Multi-PSI system. Programming of Future Generation Computers, Elsevier Science Publishers B.V. (North-Holland), 1988.

[Ueda 86] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

[Watson 87] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *Proceedings of Parallel Architectures and Languages Europe*, June 1987.