ICOT Technical Report: TR-585

TR-585

Evaluation of the KL1 Language System on the Multi-PSI

by
S. Onishi, Y. Matsumoto, K. Nakajima
& K. Taki

August, 1990

Evaluation of the KL1 Language System on the Multi-PSI

Satoshi Onishi† Yukinori Matsumoto†
Katsuto Nakajima‡ Kazuo Taki†

† Institute for New Generation Computer Technology
‡ Mitsubishi Electric Corporation

Abstract

The Multi-PSI is a loosely coupled multiprocessor, which has been developed in the FGCS project for the purpose of providing a practical tool for research and development of parallel non-numeric software. It also served as a testbed for implementation of concurrent logic language KL1 on a loosely coupled multiprocessor.

This paper reports the cost measurement of intra- and inter-processor primitive operations in the systems. They show the basic performance of our distributively implemented concurrent language. Comments for the relationship to other language systems are included. Utilization of measurement results in parallel programming are discussed from the viewpoint of reducing the inter-processor communication overhead. Measurements of performance and communication overhead in benchmark programs are also shown.

1 Introduction

The Japanese fifth generation computer project has the target of building a highly parallel inference machine (PIM) on which we construct large scale knowledge information processing systems. We have developed a prototype machine, the Multi-PSI system [Taki 88], that provides a practical tool for the research and development of parallel non-numeric software. It also serves as a testbed for an implementation of concurrent logic language KL1 [Chikayama 88] for a distributed memory architecture.

The Multi-PSI is a non-shared-memory multiprocessor, whose processing elements (PEs) are the same CPU hardware of the personal sequential inference (PSI) machine [Nakashima 87] (the microprogram is different). Up to 64 PEs are connected in an 8×8 two-dimensional mesh network with dynamic routing capability.

A distributed KL1 system was developed on the machine [Nakajima 89]. It is written in microprogram for execution efficiency. The design goal was to obtain overall high performance, taking into account garbage collection overhead, and to realize a distributed language system with a decentralized resource management mechanism for good scalability. The language system is easily expanded for a larger hardware than the current Multi-PSI with its 64 PEs.

This paper gives the cost measurement results of intra- and inter-PE primitive operations in the system, which decide the basic performance of a distributively implemented concurrent language system. Correspondence of those primitive operations to other language systems is commented on. The cost of inter-PE primitive operations gives a guideline for a programmer to control the grain size for better performance. Measurements of performance and communication overhead on benchmark programs are also shown with a discussion referring to the guideline mentioned above. Section 2 and 3 outline the

Multi-PSI hardware and the concurrent logic language KL1. Section 4 shows the measurement results of intra-PE primitive operations and comments for the correspondence of these primitives to other language systems. Section 5 overviews the implementation of the inter-PE operations and reports their cost, then discusses control of grain size to reduce the inter-PE communication overhead. Performance and communication overhead in executing parallel benchmark programs are also shown in section 6.

2 Overview of the Multi-PSI system

2.1 Hardware

The PE is a 40-bit (8-bit for tag, 32-bit for data) CISC processor controlled by horizontal micro-instruction (53 bits). The cycle time is 200 nsec. Each PE has 16 M words of local memory via a 4K-word direct-map cache memory. Address space of each PE is separated.

Up to 64 PEs are connected in an 8×8 two dimensional mesh network. Inter PE communication is done through message passing. The network has wormhole routing functionality. Each edge of the mesh includes two 8-bit channels of opposite directions. The transfer rate of each channel is 5 Mbytes/sec.

2.2 KL1 Language

KL1 (kernel language version 1) is a concurrent logic language based on Flat GHC[Ueda 86]. A KL1 program is made up of a collection of guarded horn clauses, whose form is:

$$H :- \ G_1, ..., G_m \mid B_1, ..., B_n. \qquad (m > 0, n > 0)$$

where H is called the head, G_i the guard goals, and B_i the body goals. The vertical bar ($|$) is called the commitment operator. The guard part unification is to wait for value instantiations to variables (synchronization) and to test them. When the guard unification succeeds, the control proceeds beyond the commitment bar and the body goals are executed concurrently. Those body goals may communicate with each other through their common variables.

KL1 body goals can have **pragmas** as the meta-control functions.

(1) Priority pragma (..., B@priority(Prio),...) : To specify execution priority.

(2) Throw goal pragma (..., B@processor(PE),...) : To move a goal to another PE for load distribution.

A KL1 program is compiled into KL1-B code[Kimura 87], which corresponds to WAM for Prolog, and is interpreted by a microprogram. KL1 language assumes a system-wide (global) name space. Since the Multi-PSI is a distributed memory multiprocessor, KL1 language system on the Multi-PSI requires a translation mechanism between local address space and global name (address) space, which is supported in the language implementation by microprogram.

3 Execution Mechanism of the KL1 language

Here is an overview of the execution mechanism of KL1 program in our implementation.

Goals are represented by goal records whose fields are argument slots, pointer to code, and so on. Each processor has a goal stack table which is the root of all runnable goals.

The table has pointers to goal stacks corresponding to physical priorities. The processor picks up the topmost goal of the highest non-empty goal stack and executes.

When a goal is executed, the guards of its defining clauses are tested. There are three cases:

(1) If one of them succeeds, the body part of the clause is executed;

(2) If all of them fail, a failure exception is raised;

(3) Otherwise, if none of them succeed and some of them block – that is, some of the input arguments are not sufficiently instantiated for guard test – the goal suspends on the variable(s) to be instantiated.

In case (3), a pointer to the suspended goal is written on the variable cells (the goal is said to be *hooked* onto the variables). When one of the variables becomes instantiated, the goal can be put back to the goal stack for scheduling.

The body part of a clause can contain body unification goals, body built-in predicate goals, and user-defined predicate goals. In the execution of the body part, a body unification is done *in-line*. The body built in predicate goal is also executed in-line, except when one of its input arguments is uninstantiated. In this case, a goal which execute the built-in goal is created and is hooked onto the uninstantiated argument. Goal records are allocated for the user-defined body goals and pushed onto the top of the current goal stack (thus the scheduling is depth-first), except for the last one goal which is executed tail recursively.

If the goal suspends, or succeeds but has no user-defined body goals, the next goal is picked up from the highest priority goal stack for execution.

The programmer can attach pragmas to user-defined body goals to specify execution priorities and where to move processor numbers. When a `@priority` pragma is attached to a goal, the goal is pushed onto the goal stack corresponding to the specified priority, not the previous one. When a `@processor` pragma is present, a `%throw` message is sent to the specified processor with goal information (code, arguments, priority, and so on). Only the surface level of the arguments are encoded into the message. Nested elements of lists and vectors are represented by external pointers. An external pointer is made up of a processor number and an index into the indirection table in that processor (called the *export table*). This indirection scheme was adopted, so that *local* garbage collection can be done on one processor without affecting external pointers in other processors pointing into that processor.

The value of an external pointer can be read by the `%read/%answer_value` protocol. A write to an external pointer is handled by the `%unify` protocol.

4 Intra-processor Operation Evaluation

4.1 Append Speed

An append (list concatenation) program is often used as a benchmark program for logic programming languages. An append program written in KL1 follows:

```
append([X|X1],Y,Z) :- true | Z=[X|Z1], append(X1,Y,Z1).
append([],Y,Z) :- true | Z=Y.
```

The cost of one reduction (iteration) of the first clause is 39 steps of the micro instructions in the best case (no suspension, and so on), and the speed turns out 128 KRPS (Kilo Reduction Per Second) assuming no cache miss.

In this paper, we define the cost of the above (about 8μsec) as one append-LI(Logical Inference) or one LI to normalize our measurement results in the following sections for comparing each items.

4.2 Basic Operation Costs

Figure 1 shows the costs of typical primitive operations in KL1 programs.

The cost of enqueuing a goal to the goal stack and dequeuing it is about 0.7 append-LI. As the append loop is performed in tail recursion optimization (TRO), the gain of TRO in append is $(0.7/1.7) \times 100 = 40$ %.

There are three typical cases in a guard unification: success to test an atomic data (g-1) or a structure data such as a list (g 2), suspension for non-instantiated variables (s-1 to s-4), and value mismatch, that is unification failure (this cost is the same as the matching case).

Non-busy wait mechanism is used for goal suspension. The goal is hooked to the causal variable and waits for its instantiation. `Suspension` in Figure 1 includes the sum of the costs for hooking, resuming (re-enqueuing) and dequeuing the goal (s-1). If there are two unbound variables which may allow to commit a clause, the goal is hooked to both to construct an OR-wait suspension (s-2). (s-3) is the case of an OR-wait suspension of four variables. KL1 body built-in predicates also suspend if one of their input arguments is uninstantiated (s-4).

Most body unifications in KL1 are: binding a value to an unbound variable (b-1) or making a reference pointer from an unbound variable to another (b-2). Pattern matching between atoms (b-3) or structures (b-4) are rare cases; the latter is not optimized in the current implementation.

4.3 Comments for Primitive Operation Costs

The typical intra-PE operations shown in Figure 1 can be commented as below, considering relations to other languages or systems.

Enqueue and dequeue cost (e-1) corresponds to process fork and scheduling cost in other languages or systems. In a KL1 program, body goals, which appear in a clause definition, specify process fork. Each body goal becomes a process when the clause committed. A process can have wide range of grain size. When the grain size is very small, like append of one element lists, the enqueue and dequeue cost cannot be ignored. But when the grain size is larger than seven append-LI, enqueue and dequeue cost affect the performance much less. We call this grain size "fork grain size" here.

Single suspension cost (s-1) corresponds to a synchronization cost. Assuming two processes, one of which is a producer of a value for a shared variable and another is a consumer, the consumer has to be suspended when it touches the shared variable until the variable has a value. When the producer passes a value sequence to the consumer through a stream, like a pipelining, synchronization has to occur repeatedly. The interval of the synchronization decides how much the suspension cost affects total performance. That is, when suspension occurs every 1.8 append-LI in a process execution, the suspension cost (1.8 append-LI) reduces the process execution performance to 50 %. We consider the other grain size here. We call the interval of synchronization "synchronization grain size".

A programmer has to control grain size in the program to get good performance. Measurements like (e-1) and (s-1) give guidelines for a programmer to control the lower bound of the fork grain size and synchronization grain size in the program.

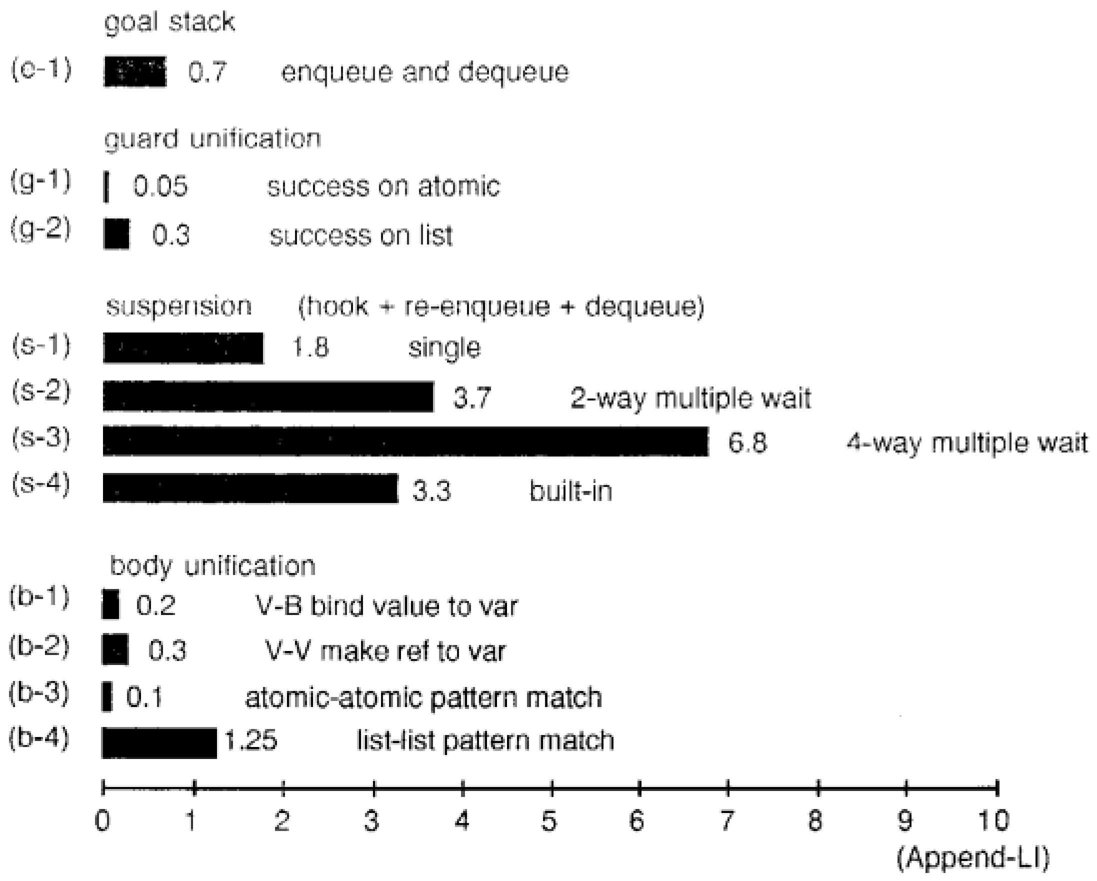

Figure 1: Cost of Typical Intra-PE Operation

5 Inter-processor Operation Evaluation

The KL1 distributed implementation was designed for distributed memory parallel machines, and there are many mechanisms to minimize the number of inter-PE messages. However, this policy often increases the message handling cost. This section overviews the inter PE mechanisms of goals, data, and pointers passing or management (more details are in [Nakajima 89]), and shows the costs for handling typical messages. Inter-PE communication overhead and relations to the grain size are also discussed, referring the message handling cost.

5.1 Inter-processor Operation In KL1

A KL1 goal with a throw goal pragma is shipped out to another PE by the **%throw**[1] message. As the single assignment semantics of KL1 allows data copying, the atomic arguments of the goal are copied into the message. However, if the argument is a structure, it is encoded as an *external reference pointer* instead of copying it, because the goal might not need the data in that PE. The structure elements are transferred lazily on request by **%read** messages. To generate an external pointer is called *exporting*, and to receive the external pointer is called *importing*. Unbound variables are always exported as external pointers to retain their identity of the variables in the system.

To perform local GC, we separated address space into two layers: intra-PE address space and inter-PE(global) address space (name space). All the exported data must be known by the PE so that they are not reclaimed as garbage. For this purpose, the **export table** keeps the internal addresses of all the exported data in a PE. All the external pointers point to the entries of the table from outside the PE. They are represented in the form $< n, e >$, where n is the exporting PE number and e is the entry position in the export table. This address is not affected (changed) by local GCs.

The entries of the export table are managed and incrementally reclaimed by a weighted reference counting scheme called WEC [Ichiyoshi 88]. In this scheme, a weighted count of a positive integer, called WEC, is kept on both the export and import sides. A certain amount of weight is attached to every exportation of pointers. The exported weight is accumulated at the exportation side as a negative value, and the imported weight is accumulated at the importation side as a positive value, corresponding to each variable. When a pointer is consumed, the accumulated imported weight is returned to the exportation PE. This scheme reduces number of messages much better than the full reference counting. To keep WEC on the importing side, we have the *import table*.

On exporting and importing pointers, it is necessary to translate its address from/to the inter-PE one and to handle WEC. As the Multi-PSI has no special hardware for these operations, they are performed by the microprogram.

5.2 Message Handling Costs

Figure 2 shows the costs for handling typical messages. The measurement condition follows:

- The costs of sending and receiving a 65-byte **%throw** message whose three arguments are an atom and two external pointers as in a typical situation.
- The costs of sending and receiving a 14-byte **%read** message which request the contents of an external pointer, and a 24-byte **%answer_value** message which answer

[1]Inter-PE messages, %throw, %read, and so on, are generated by KL1 language system, not handled by programmers.

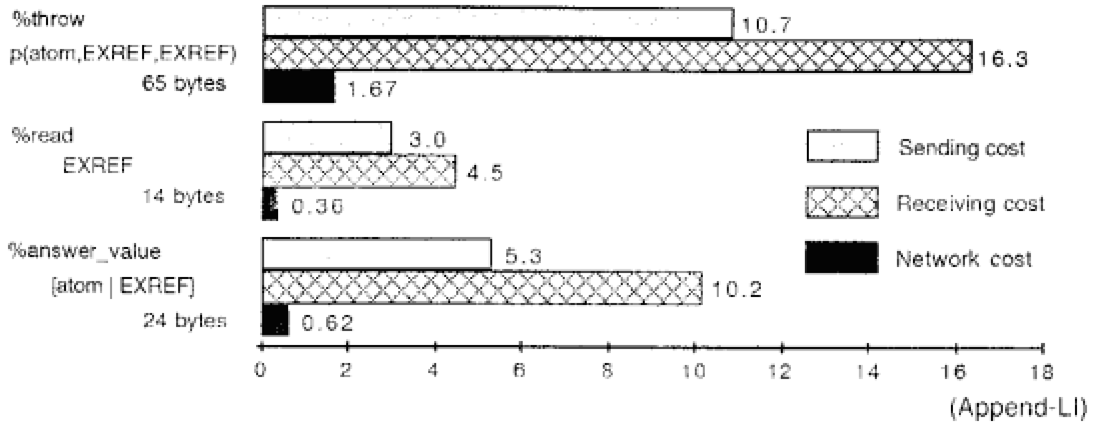

Figure 2: Cost of Typical Inter-PE Operation

the request. The returned data is a list whose CAR is an atomic data and the CDR is an external pointer.

The routing hardware has 5M bytes/sec of the bandwidth for transmitting the message. Compared with the network cost (hardware capability: 1.67/0.36/0.62 append-LI for 65/14/24-byte), the sending and receiving cost of the microprogram execution is quite large. It includes the cost of address translation, encoding and decoding messages, and distributed goal management and other resources management.

5.3 Discussion

Let's see the typical inter-PE operation in Figure 2, making an analogy to the discussion in section 3.3.

`%throw` operation corresponds to a process fork to a different processor. The cost of the `%throw` operation gives a guideline for a programmer to control the lower bound of the fork grain size thrown to a different processor.

`%read` and `%answer_value` operation correspond to a synchronization with a variable placed in a different processor. The cost of these operations gives a guideline to control lower bound of the synchronization grain size across a processor boundary.

Generally speaking, as the number of grains becomes larger, the load balancing becomes easier. In this case, the grain size tends to become smaller. A programmer divides his problem into smaller grains and distributes them among processors to get better load balance. But when the grain size becomes too small, the cost of `%throw`, or `%read` and `%answer_value` affect overall performance significantly. It means that a programmer has to keep the grain size larger than a certain size, which can be calculated based on the `%throw` cost or `%read` and `%answer_value` cost.

Let's see an example of deciding a lower bound of grain size. We assume a program in which only the synchronization grain size affects performance. Inter-PE synchronization cost, a summation of `%read` and `%answer_value` cost, is 23 append-LI. When a programmer accepts 50% performance down caused by those costs, the lower bound of the synchronization grain size becomes 23 append-LI. This means that interval of the inter-PE synchronization should be larger than 23 append-LI to keep the performance degradation below 50%.

There is an open problem whether the inter-PE operation cost shown in figure 2 is too large or not, compared with the intra-PE performance. We expect that the cost is reasonable. That is, the same order of implementation cost is always required when a language system with a global name space is implemented on a distributed memory machine.

6 Measurements on Benchmark Programs

We took measurements for two different types of benchmark programs.

- **Pentomino:** A program to find out all solutions of a 5 × 8 packing piece puzzle. That is, it finds all the ways of packing ten variously shaped pieces into a 5 × 8 rectangular box (to be precise, it is the tetromino, a smaller version of pentomino).
- **Bestpath:** A 160 × 160 grid graph is given together with randomly generated non-negative edge costs. A program determines the lowest cost path from one vertex to all other vertices of the graph (single-source shortest path problem).

The Pentomino program does an exhaustive search of an OR-tree of possible piece placements. It runs at 39.3KRPS on 1PE. The cost of one reduction is 3.3 append-LI. It runs 8.34 Mega reductions in 4.3 sec on 64 PEs, and totally it runs 1.9 MRPS (this is the top speed in the world). The dynamic load balancing scheme in [Furuichi 90] works well, and as shown in Figure 3, 50-fold speed-up on 64 PEs is attained, which is almost linear speed up. The inter-PE fork grain size of this program is 88 pentomino LI and Figure 2 shows that the grain size is much bigger than %throw cost. There is almost no communication between sub-trees of the OR-search, which means that only the fork grain size affects total performance in this program. Since the fork grain size is big enough, as mentioned, the communication overhead is small as measured in Figure 3.

The Bestpath program generates 160 × 160 processes corresponding to every grid, then performs a fully distributed shortest path algorithm. In the algorithm, adjacent processes exchange messages to search shortest paths. It runs at 23.4KRPS on 1PE. The cost of one reduction is 5.5 append-LI. Grids are separated into many groups[2] and statically assigned to PEs. This program runs 1.5Mega reductions in 1.7 sec on 64 PEs, and its performance is 0.89 MRPS. Inter-PE synchronization grain size is about 12 bestpath-LI, and rate between grain size and message handling cost in Figure 2 is not larger than rate of pentomino's, so Figure 3 shows that communication overhead is large. Figure 3 shows that the communication overhead and the cache miss penalty degrades the performance by 40 % on 64 PEs, though the idle rate is smaller than that of Pentomino.

Both fork grain size on pentomino and synchronization grain size on bestpath in a PE are very small, they are 1 to 2 reductions. The problem of the load mapping in KL1 is how to make large inter-PE grains from many small grains and how to map to each PE. It is different from dividing a big problem into smaller pieces.

7 Conclusion

The Multi-PSI is a prototype parallel inference machine used as a testbed for researching on parallel language implementation. KL1 was implemented on it with much effort given to (1) minimizing the overhead of garbage collection, (2) reducing the number of inter-PE messages, and (3) distributing information, so that the system might be efficient and scalable.

This paper gave the measurements of intra- and inter-PE operations in the system.

[2] 16 times as many as the number of PEs

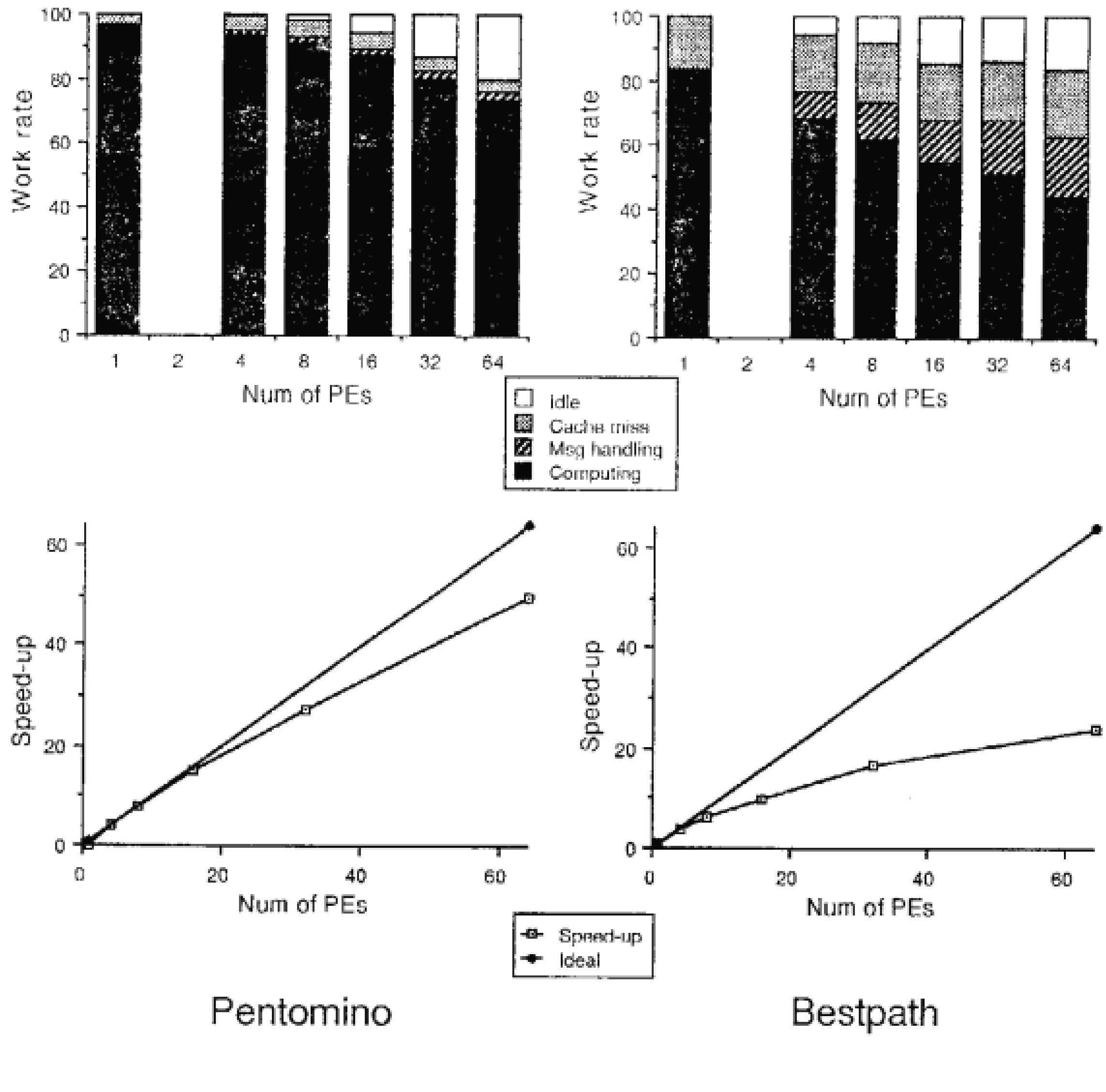

Figure 3: Runtime Analysis and Speed-up

The peak performance on 1 PE is 128 KRPS and is at the highest level among the current distributed machines. Typical guard and body unifications are performed very quickly by utilizing the tag architecture. The suspension costs are not small because the PE does not have any special hardware for manipulating goal records. Inter-PE message handling costs are 3 to 16 append-LI. Most of the time for them is spent on micro code execution and the network speed does not limit the inter-PE communication performance.

We also discussed the fork and synchronization grain size and consideration of their lower bound in KL1 programming. Programmers can expect that inter-PE communication overheads are small enough when these grain sizes are bigger than the lower bound decided by the inter-PE primitive operation costs.

The dynamic characteristics of two parallel benchmark programs show that the inter-PE communication overhead can be limited and a good workrate can be attained by investigating well-organized dynamic or static load balancing and appropriate grain size control.

Our insight into loosely-coupled multiprocessors and parallel programming on them is still very limited. We will continue to experiment with more programs with different runtime characteristics, and conduct more detailed measurements and analysis.

References

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

[Furuichi 90] M. Furuichi, N. Ichiyoshi and K. Taki. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1990.

[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.

[Kimura 87] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of 1987 Symposium on Logic Programming*, Sept. 1987.

[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.

[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. Also in ICOT Technical Report TR-237 and in Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium 1987.

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