TR-466

Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2

by

Y. Inamura, N. Ichiyoshi, K. Rokusawa
& K. Nakajima

April, 1989

Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2

Yū Inamura Nobuyuki Ichiyoshi
Kazuaki Rokusawa Katsuto Nakajima
Institute for New Generation Computer Technology

Abstract

The Multi-PSI/V2, a loosely coupled multiprocessor running the parallel logic programming language KL1 (kernel language version 1), has been developed for conducting parallel software research and for testing various new implementation techniques.

From the outset of the design of the KL1 implementation on the machine, attention has focused on the realization of efficient memory management, which is one of the serious problems of a committed-choice language implementation. The new multiple reference bit (MRB) technique provides a way of performing incremental local garbage collection efficiently. It turns out that the MRB information can be used also for destructive update of structure elements, speedup of the built-in constant-time merger, and efficient inter processor data management. This paper describes these optimization techniques. Measurement results are also given, and they confirm the effectiveness of the optimization.

1 Introduction

The Multi-PSI/V2 [6, 7] is a loosely-coupled multiprocessor developed in the Japanese fifth generation computer system (FGCS) project. The purpose of development was (1) to provide a system powerful enough to run large-scale parallel logic programs before the Parallel Inference Machine (PIM) [4], one of the targets of the project, is available, and (2) to test various new implementation techniques for KL1. The initial implementation of the parallel inference machine operating system (PIMOS) [2] has been completed and some medium-scale programs are running. Performance measurements are also being conducted.

In designing the KL1 implementation on the Multi-PSI/V2, much attention has been paid to the memory management scheme. This is because the heap-based execution of KL1 consumes memory quite rapidly, and it is expected that a naively implemented memory management scheme will become a limiting factor of the overall system throughput.

The multiple reference bit (MRB) scheme [1] provides a way to perform incremental garbage collection. It turns out that the MRB information can be used not only for incremental garbage collection but also for various other optimization techniques related to memory management. This paper describes these optimization techniques, and gives the results of the performance measurements. The results confirmed the effectiveness of the optimization techniques, and proved that the wide applicability of the MRB mechanism.

2 The MRB Scheme

The MRB is one-bit information attached to every reference pointer. The implementation maintains the MRB so that if the MRB of the pointer is off, the pointer is guaranteed to be the sole pointer to the data. When a pointer with the MRB off is consumed — i.e. the referenced data is read and the pointer is no longer needed, or the pointer is simply discarded —, the memory area occupied by the data can be reclaimed because it becomes inaccessible from the program.

As for variables, one can have up to two pointers with the MRB off. Typically, a variable is shared by a producer process and a consumer process, and is instantiated to a concrete data via one of the pointers, and the variable cell together with the data will be reclaimed when the data is read via the other pointer.

It can be decided at compilation which pointers are consumed or duplicated in a reduction using a given clause. The compiler generates instructions for reclaiming garbage data or instructions for turning the MRB of the pointer on. One of the merits of the MRB technique is that no extra memory accesses are needed except when the referenced data is garbage collected by the scheme. This will be all the more important in a shared-memory multiprocessor like the PIM, where memory access contentions should be kept as infrequent as possible.

It is known that, for a wide class of symbolic processing programs, the majority of data is single-referenced. KL1 programs are no exception: we have found 40 to 90 percent of garbage data is incrementally reclaimed by the MRB scheme in benchmark programs.

The reader is referred to [1] for details of the MRB scheme.

3 Optimization with MRB

3.1 Destructive Update of Structures

In logic programming languages like KL1, only variable data can be assigned a value, and data instantiated once can never be modified logically. Therefore, when structure data which is almost the same as an existing structure except for a few elements is needed, it is necessary to create a new structure by copying elements from the old one to the new one. However, with the MRB mechanism, which gives information about the existence of other pointers, a structure can be reused by updating some elements destructively.

The following are examples in which a structure can be reused when its MRB is off.

foo1([X1|X2]) :- true | bar([Y1|Y2]), ...

foo2([X1|X2]) :- true | bar([X1|Y2]), ...

In the first example, the cons cell is reused (structure frame reuse); in the second example, the cons cell and CAR of the list are reused (structure element reuse).

3.2 Inter-PE Data Management

An *external reference*, a reference which points to a data object in another PE may be created as a result of goal distribution. The goal sender PE *exports* the data object and the goal receiver PE *imports* it. For local garbage collection, an external reference points to a data object through two indirection records called an *import table entry* and an *export table entry*, which are never moved in local garbage collection [5]. The hashing mechanism was introduced in exportation and importation, to avoid the multiple registration of one data object, which may increase the number of inter-PE messages.

```
merge([], In2, Out) :- true | Out = In2.
merge(In1, [], Out) :- true | Out = In1.
merge([X|In1], In2, Out) :- true | Out = [X|Out2], merger(In1,In2,Out2).
merge(In1, [X|In2], Out) :- true | Out = [X|Out2], merger(In1,In2,Out2).
```

Figure 1: Definition of "merge/3"

Import and export tables with simpler structures, called *white import and export tables*, were also introduced, and used for data export and import where the MRB is off [6]. The original import or export table is called the *black import or export table*, in contrast with the new table. The hashing mechanism is not necessary for the white export or import, because in most cases, the sole pointer to the object is exported and there are no more internal pointers to the object. The data object is never re-exported. The procedures of export and import are fairly simplified by white export and import table.

3.3 Stream Merger

Stream communication between processes is a heavily-used programming technique in committed-choice languages. When a process may receive messages from an unknown number of processes, message streams of the sender processes must be merged into one stream to the receiving process. This is a common situation in a object-oriented style program, in which a reference to an object (input stream to the process representing the object) may be duplicated at any time.

A two-way stream merger can be defined in KL1 as shown Figure 1.

The elements of two input streams are merged and sent to the output stream by this process. However, there are two obvious disadvantages in such a realization of the stream merge operation:

- Suspension and resumption of the "merge/3" process are necessary for each element of the input stream, and these are fairly expensive procedures;
- The cost of merge operation increases in proportion to the increase in the number of the input streams.

One sophisticated solution to the problem was proposed, using a special structure to describe predicates[8]. The solution was general and able to applied to the predicates other than merge, as long as they have no guard. We introduced a more specialized solution which is only applicable to the merge process.

This is done by preparing special expressions for the variable for which a merger process is waiting (merger hooked variable: MHV), and for the merger process itself (merger record: MR). The MHV points to the MR and the MR points to a variable cell which corresponds to the output variable of the merger process. The MR also contains an integer value which indicates the number of input streams to itself. This value is used to detect the termination of a merger process (Figure 2).

A unification between an MHV and list data causes the instantiation of the output variable with new list data immediately, thus eliminating the overheads associated with suspension in the user-defined merger (Figure 3).

The built-in merger can be further optimized. When the input cons cell is single-referenced, it can be reused as the new output cons cell.

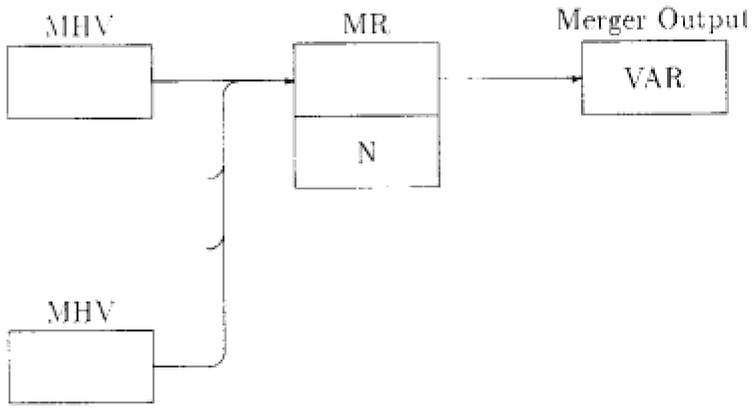

Figure 2: Representation of merger process

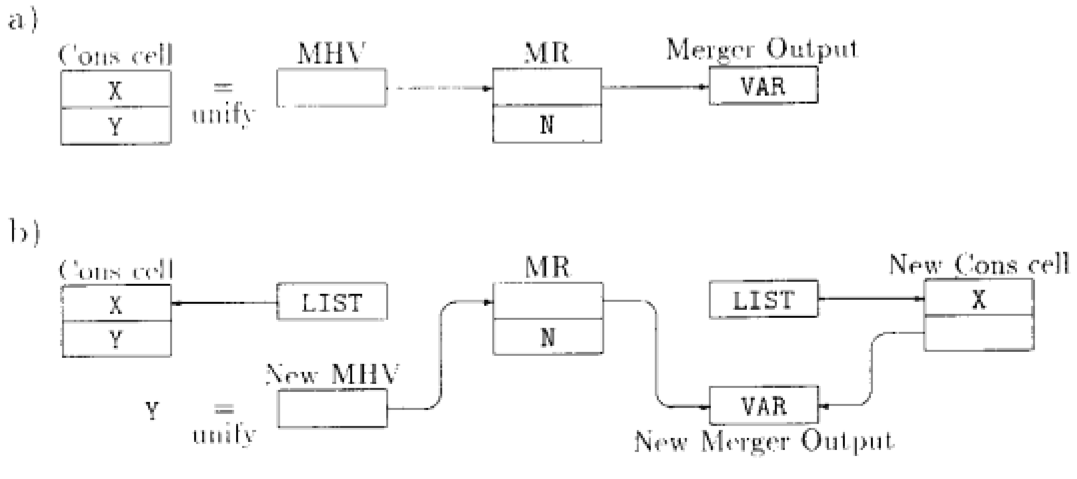

Figure 3: Unification between an MHV and list data

```
merge([],     Out) :- true | Out = [].
merge([A|In],Out) :- true | Out = [A|NewOut], merge(I,NewOut).
merge([],     In,     Out) :- true | merge(In,Out).
merge(In,     [],     Out) :- true | merge(In,Out).
merge([A|I1],I2,      Out) :- true | Out = [A|NewOut], merge(I1,I2,NewOut).
merge(I1,     [A|I2],Out) :- true | Out = [A|NewOut], merge(I1,I2,NewOut).
merge([A|I1],I2,      I3,     Out) :- true |
                                Out = [A|NewOut], merge(I1,I2,I3,NewOut).
merge(I1,     [A|I2],I3,      Out) :- true |
                                Out = [A|NewOut], merge(I1,I2,I3,NewOut).
merge(I1,     I2,     [A|I3],Out) :- true |
                                Out = [A|NewOut], merge(I1,I2,I3,NewOut).
                :
                :
merge({},      Out) :- true | Out = [].
merge({X},     Out) :- true | merge(X,Out).
merge({X,Y},   Out) :- true | merge(X,Y,Out).
merge({X,Y,Z},Out) :- true | merge(X,Y,Z,Out).
                :
                :
merge({},    In,    Out) :- true | merge(In,Out).
merge(In,    {},    Out) :- true | merge(In,Out).
merge({X},   In,    Out) :- true | merge(X,In,Out).
merge(In,    {X},   Out) :- true | merge(In,X,Out).
merge({X,Y},In,    Out) :- true | merge(X,Y,In,Out).
merge(In,    {X,Y},Out) :- true | merge(In,X,Y,Out).
                :
                :
```

Figure 4: Logical definition of the stream merger

Another important requirement for the merge process is the dynamic increase of the number of input streams. With the KL1 definition of the merger, it is realized by nested merge processes, which needs very expensive procedure. Our solution is to expand unification of the MHV from the original definition of merge (Figure 4. The unification between an MHV and vector data is accepted, which is regarded as the addition of input streams. When a vector is unified to an MHV, as many input streams (MHVs) as the arity of the vector are added to the merger process and each element of the vector is unified to each new input stream.

By expanding the unification like above, the merger process can be regarded logically as a set of an infinite number of clauses which have two to an infinite number of arguments as shown in Figure 4.

For simplicity, only "merge/2" is prepared as a built-in predicate, and merger processes which have multiple input streams are realized only by the unification between an MHV and a vector.

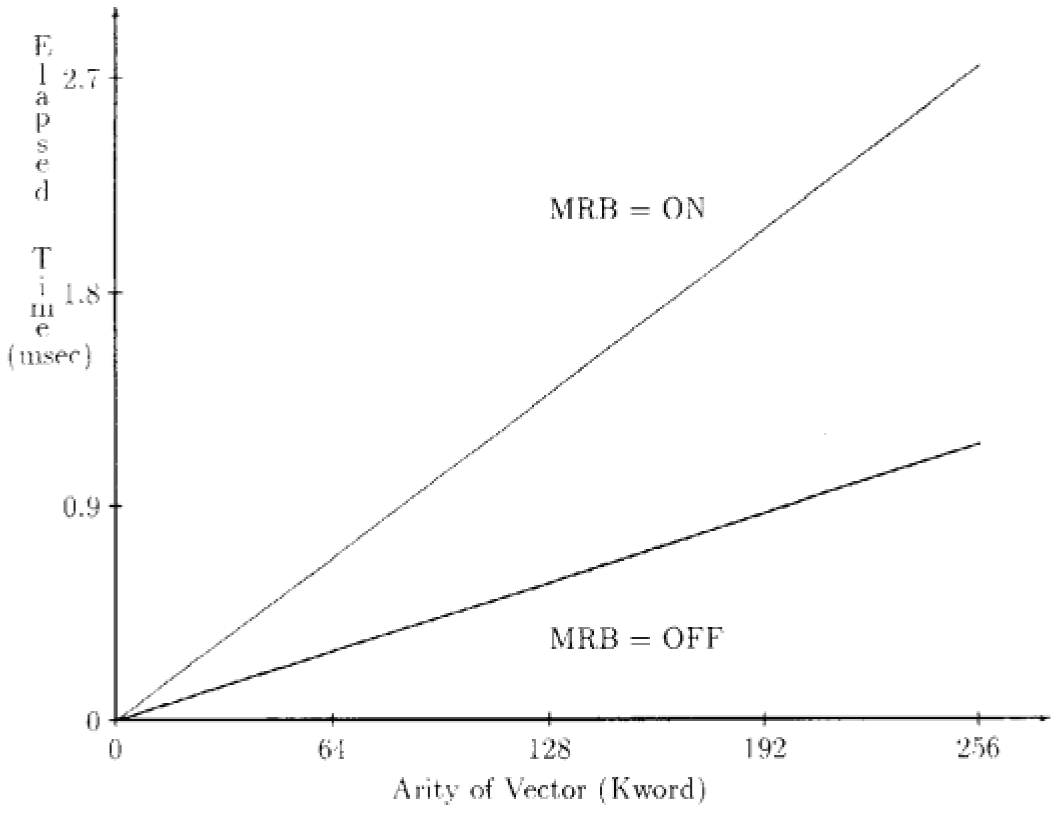

Figure 5: Update cost of vector (modifying all elements)

4 Evaluation of Optimization

This section shows the results of performance measurements of the optimization techniques described in the previous section. For each item, we took the processing time for data with MRB on and off. The former corresponds to the processing time without MRB optimization. The time difference represents the effectiveness of the particular optimization.

4.1 Structure Reuse

4.1.1 Vector Update

To ascertain the effect of the MRB, the modification costs of vectors with several size were measured. The measurement program modified all the elements of the given vector with the MRB on or off for the comparison, and the time elapsed in the whole procedure was measured.

As the mutable array scheme [3] was introduced to deal with the bad case in updating vector, the modification cost of one element is constant even if the vector's MRB is on. However, it becomes obvious that the modification cost of a vector with MRB on is about 2.5 times as much as that of a vector with MRB off.

4.1.2 Effect in Small Bench-mark Programs

The effect of the structure reuse was measured with several small bench-mark programs, such as *Append*, *Quick-sort*, and *Prime number generator*. Each program was compiled in three ways and the execution speed of each is measured. The three ways are:

1. Without any structure reuse;
2. With structure frame reuse;
3. With structure element reuse.

Table 1 shows measurement result.

Table 1: Performance improvement with structure reuse

	No reuse (KRPS)	Frame reuse (KRPS)	Element reuse (KRPS)
Append	110	128	146
Qsort	95.8	108	113
Primes	55.9	59.8	61.2

RPS: reductions per second

The performance improves by 10% to 30% with structure element reuse in these small bench-marks. This improvement is mainly brought about by the decrease of the number of instruction steps when running these small bench-marks. However, structure reuse can also reduce the frequency of memory access, and this may greatly affect the performance with shared memory machines such as the *parallel inference machine* (PIM) [4].

4.2 Stream Merger

The cost of the built-in merge procedure is compared with the merger defined with KL1 (Figure 1), to find out how the built in merger improves the performance.

Figure 6 shows the cost of merging one element, with the input list's MRB both on and off.

The difference between the KL1 merger and the built-in merger with the MRB on can be regarded as the effect of the built-in merger representation, which can be realized even in the implementation without the MRB mechanism. The performance improved by four times with the introduction of the built-in merger.

The difference between the built-in merger with the MRB off and on is the effect of the MRB. Execution with the MRB off is twice as fast as that with it on.

Although the performance improvement with the built-in merger seems to be sufficient, we are convinced that MRB mechanism is necessary, since merge operation is used quite frequently in KL1 programs, and influences the overall system performance.

4.3 Comparison of Black and White Exportation

To ascertain the effect of introducing the white export and import table, the cost of goal distribution and data transfer were measured, using white exportation and black exportation independently.

4.3.1 Cost of Goal Distribution

The costs of goal distribution with several arities were measured. Three kinds of arguments are thrown with the goal:

1. Variable with the MRB off;
2. Variable with the MRB on;

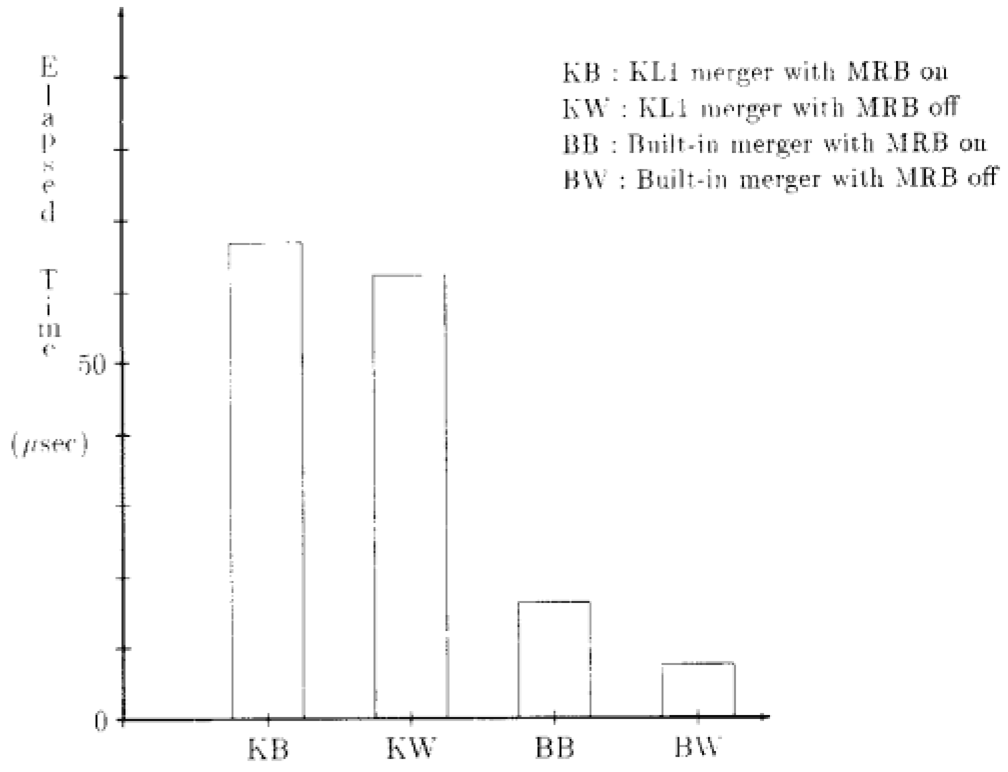

Figure 6: Performance of the merge operation

3. Atomic data.

The first kind of argument is exported with a white export table entry. The second is exported with a black export table entry. The third is exported as is. The time elapsed in one goal reduction in each case is shown in figure 7

Assuming that the cost of exporting atomic data is a constant overhead in the goal throw operation, the cost of black exports is about three times as expensive as the cost of white exports. However, as a loosely-coupled multiprocessor, the constant overhead in the goal throw operation is very large. Thus, the difference between the white and black exportation is not significant, in particular, when the arity of the thrown goal is little. As for the goal throw, it can be said that the white export table is an optimization for the large goal distribution.

4.3.2 Cost of Data Transfer

The data of an external reference is transferred by the inter PE messages **read** and **answer_value**. The costs of data transfer using this protocol are measured in terms of the white and black exportation. The data transferred is lists with several sizes. In our implementation, only one cons cell is transferred at one time and the CDR of the cons cell is represented by an external reference if it is pointed to another cons cell, because of the on-demand copy strategy [6]. The cost of one cons cell transfer is shown in Table 2.

Even in transferring one cons cell, the white export reduces the communication cost by 20%. This is because the constant overhead of the **read** and **answer_value** is much less than that of the throw goal operation. White export table is fairly effective in the case of data transfer.

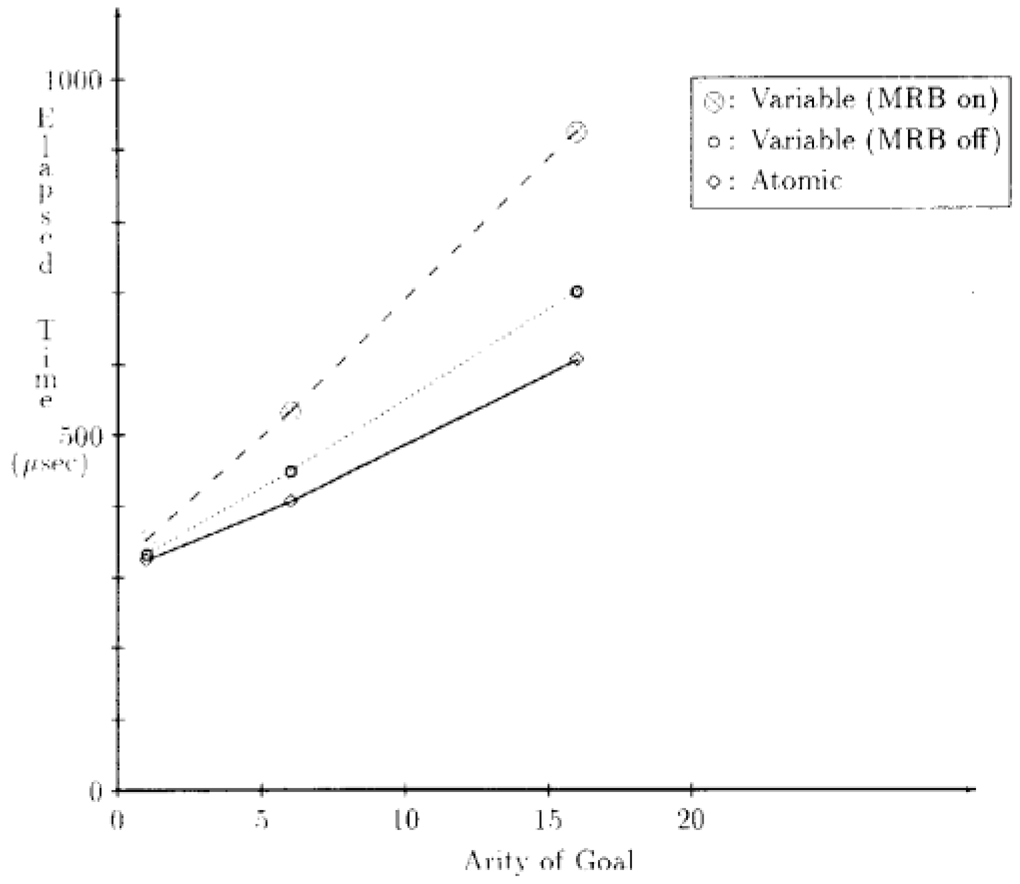

Figure 7: Elapsed time in one goal reduction

Table 2: Cost of one cons cell transfer

White export	Black export
240 μsec	300 μsec

5 Conclusion

This paper described several optimization techniques using the MRB information, such as *structure reuse*, *built-in constant-time stream merger*, and *white export*. The effects of these optimization techniques were individually measured, and it was ascertained that these optimization techniques improved the respective performances by 10% to over 200%.

This proved that the MRB mechanism not only makes incremental local garbage collection possible but also enables speedups of various typical processing related to memory management.

Although the measurement results confirmed the effectiveness of each piece of optimization, the impacts on the overall performance of the system should be evaluated by measuring the frequencies of optimized operations. We are taking those measurements using realistic-size programs of various programming styles.

Acknowledgments

We would like to thank the ICOT Director, Dr. K. Fuchi, and the chief of the fourth research laboratory, Dr. S. Uchida, for giving us the opportunity to conduct this research. We would also like to thank the researchers of ICOT and the cooperating companies, who have worked with us in designing, implementing, and evaluating the KL1 system on the Multi-PSI/V2.

References

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

[2] 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.

[3] L. H. Eriksson and M. Rayner. Incorporating mutable arrays into logic programming. In *Proceedings of the Fourth International Conference on Logic Programming*, Upssala, 1984.

[4] 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.

[5] 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.

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

[7] 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.

[8] K. Ueda, and T. Chikayama. Efficient stream/array processing in logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1984.