ICOT Technical Report: TR-338

TR-338

Lazy Reference Counting Method-An Incremental Garbage Collection Method for Parallel Inference Machines-

by
A. Goto, Y. Kimura
T. Nakagawa & T. Chikayama

Febrary, 1988

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

Lazy Reference Counting Method

– An Incremental Garbage Collection Method for Parallel Inference Machines –

Atsuhiro GOTO* Yasunori KIMURA Takayuki NAKAGAWA
Takashi CHIKAYAMA

Institute for New Generation Computer Technology (ICOT) †

Abstract

Garbage collection (GC) plays an important role in attaining efficiency in implementations of parallel logic programming languages. In addition, not only normal execution but also GC must be executed in parallel with good memory access locality. Otherwise, the performance of the total system will be seriously damaged by the GC overhead.

The lazy reference counting (LRC) method, proposed in this paper, is a kind of reference counting GC. LRC uses the fact that there are few multi-referenced data objects. Not every data cell has a reference count field. To indicate multiple reference, an indirect pointer of two-words, called the *RC cell*, is used in LRC. One word in an RC cell is used to point a multi-referenced data object, and the other is used to show the number of references to the data object. Pointers to the RC cell have a special tag, a pointer tag, to show they are pointing an RC cell. The abstract machine instruction set includes instructions which maintain the number of references, and collect garbage cells incrementally during execution.

LRC GC is suitable for implementing parallel logic programming languages on parallel processors with shared memory and local coherent cache memory. This is because LRC operations have small execution overhead, and their memory operations have high locality. LRC uses one additional tag type and extra memory only for RC cells, but it does not require special memory hardware such as structure memory in data flow machines.

1 Introduction

The parallel inference machine (PIM) is one of the most important research target of the FGCS project [8,9]. The principal aim of parallel processing is to increase the execution performance so that users will be able to solve large application programs. Efficient memory management is very important in the PIM because the garbage collection performance is critical to such AI-oriented systems.

The target language of PIM is a parallel logic programming language, KL1 [8], which is designed based on GHC [17,18]. KL1 is a committed choice logic programming language [4,5,15,17,18] without backtracking. KL1 can describe the basic operations such as synchronization and communication between parallel processes without side-effects.

Naive implementations of such parallel logic programming languages consume memory area very rapidly. For example, whole array elements must be simply copied when only one element is updated because destructive assignment is not allowed. As a result, garbage collections (GC) will occur frequently. In addition, the locality of memory references is not good during GC by widely used methods, so that cache misses and memory faults will occur often. In sequential Prolog [20], this problem is not very serious because of the backtracking feature.

*CSNET: goto%icot.jp@relay.cs.net, ARPA: goto%icot.uucp@eddie.mit.edu, UUCP: ihnp4!kddlab!icot!goto
†Mitakokusai Building 21F, 4-28, Mita 1, Minato-ku, Tokyo, 108, JAPAN

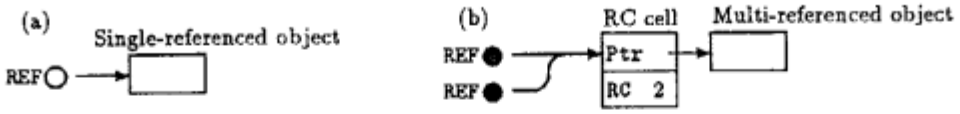

Figure 1: Multiple reference by RC cell

However, as other committed choice languages have no backtracking, an efficient incremental garbage collection method is important in their implementation.

Reference counting [6,10] is one method to incrementally recognize when a certain storage area has become inaccessible from the program. This method is commonly used in parallel processor systems such as dataflow machines [19]. However, there are two major problems in reference counting [6,7]:

- In principle, each word cell must have a reference counter field for the whole memory space.
- The cost of updating the reference counter is high, because data objects must always be accessed.

Several methods were proposed to reduce these overheads relying on the fact that data objects are not used very many times, and most are used only once. Multiple reference bit (MRB) method [2] was proposed as an incremental garbage collection method for committed choice logic programming languages[1]. The MRB method has many advantages. However, as the MRB method uses only one-bit information to show multiple references, it cannot reclaim the garbage cells which once had multi-referenced. Therefore, we must augment the MRB method with conventional garbage collection. In Lisp, Deutsch and Bobrow [7] proposed to keep a hash table[2] for the reference count of multi-referenced cells.

This article proposes an incremental garbage collection method called lazy reference counting (LRC). LRC introduces two-word indirect pointers with a reference counter, instead of a hash table in [7]. LRC makes up for the deficiency of the MRB method because LRC can reclaim storage area that are no longer used, while keeping most advantages of the MRB method. This paper presents the representation and maintenance of LRC along with an abstract machine for KL1 and its instruction set augmented with this feature.

2 Lazy Reference Counting Method

2.1 Indirect pointer with reference count

LRC is a kind of reference counting garbage collection, reducing the cost of reference count maintenance and the memory space for reference counts. LRC does not provide a reference count field in the data objects themselves, because most data objects can be expected to be used only once. When a data object obtains two or more references, an indirect pointer cell with reference count will be allocated lazily, as shown in Figure 1. The indirect pointer cell is called an *RC cell*. Therefore, reference pointers meet only at the RC cells. The first field of an RC cell is used to represent a pointer to a multi-referenced data object[3], the other is used to represent the reference count, i.e. the number of pointers meeting at this RC cell.

2.2 Multiple reference pointers

Pointers to an RC cell are called multiple reference pointers. They are indicated by a special tag[4], and in this article, with a black circle, REF ● , shown in Figure 1. All RC cells are pointed only by pointers of REF ● . There are two kinds of pointers in KL1 implementation, structure pointers pointing list cells or arrays, and indirect pointers to bind unbound variable cells. They can be classified as follows:

[1] [3] shows the statistical feature in Lisp.

[2] The multireference table (MRT). The zero count table (ZCT) is used to keep free cells.

[3] Atomic values or unbound variables in KL1 can be placed inside RC cells.

[4] This tag corresponds to the multiple reference bit in the MRB method [2].

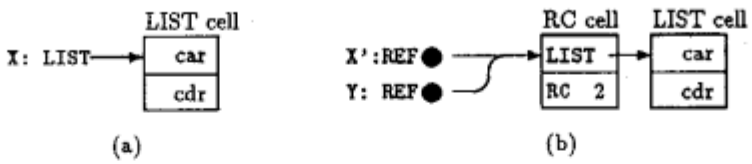

Figure 2: Insertion of an RC cell

REF● (multiple reference pointer): The pointed cell is an RC cell. Therefore, this is one of multiple reference pointers to the cell.

REF○ (single reference pointer): The pointed cell is not an RC cell. This is the only pointer to the cell[5].

LIST,VECT (structure pointers): The data object pointed by these pointers is the body of a structure such as a list cell or an array. These are the only pointer to the data object.

2.3 Update and lazy allocation of RC cells

When a reference path to a multi-referenced data object is consumed or generated in the execution of programs, the reference count of the RC cell that appeared first in the reference path is updated. If there is no RC cell in the reference path, the data object is a single-referenced data object (Figure 2(a)). When such a reference path is consumed, the storage area for the reference path and the data object can be reclaimed. On the other hand, when an additional reference path is created to the single-referenced data object, a new RC cell is inserted just before the data object, as in Figure 2(b). This operation is called *lazy* allocation of an RC cell.

3 Maintenance of Reference in KL1

Reference paths to data objects are created and consumed in the following operations during execution of KL1[6] programs.

3.1 Creation of data structures

In the execution of KL1, data structures are only created as arguments of body goals[7].

```
p :- true | q([X|Y]).
```

Reduction by this clause will create a new list cell, `[X|Y]`. When a new structure is created as an argument of a body goal, there can be no other reference paths to the structure. Therefore, no RC cell is required in the reference path to this structure.

3.2 Creation of new variable cells

Because of the single assignment nature of KL1, an unbound variable cell usually has one reference path for instantiating and one or more reference paths for referencing its value[8]. Therefore, an unbound variable cell with only two reference paths is represented without an RC cell, as in Figure 3(a), and one with more than two reference paths is represented with reference count, as in Figure 3(b). The latter is called an *RC variable cell*.

Creation of a new variable cell is required when body goals have a variable which is not passed from the reduced goal, as in:

[5] If the pointed cell is an unbound variable, there is another pointer, as mentioned in section 3.2.

[6] In the following explanation, KL1 is almost comparable to flat GHC [17,16].

[7] Creation of new data objects is never required by guard part unification in KL1.

[8] An unbound variable cell with only one reference path is called a *void* variable, which usually means *don't care*.

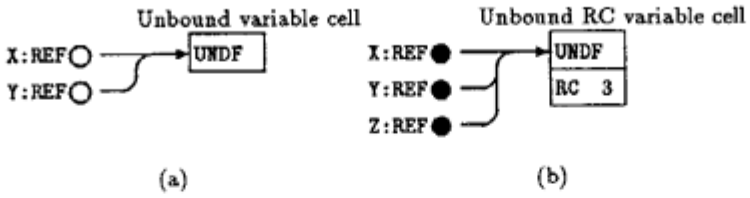

Figure 3: Representation of unbound variable cells

```
foo :- true | p(X), q(X).          (1)
bar :- true | p(Y), q(Y), r(Y).    (2)
```

Reduction by clause (1) will create a new variable cell for X, and clause (2) for Y. When a new variable cell is created, there can be no other reference paths to the newly created variable cell except those in the clause. Thus, representation of a new variable cell can be determined by how many times the variable appears in the body. Therefore, the variable cell for X in clause (1) should be represented as in Figure 3(a), and for Y in clause (2) as shown in Figure 3(b).

3.3 Passing variables between KL1 goals

Consider the following two examples:

```
p(X) :- true | q(X).          (1)
p(Y) :- true | q(Y), r(Y).    (2)
```

Variable X in clause (1) will simply be passed from the reduced goal, p, to the next goal, q. In this case, the number of reference paths to X will be kept unchanged by a goal reduction using this clause.

On the other hand, variable Y in clause (2) will be duplicated in body goals q and r. In this case, the number of reference paths to Y will increase. When the data object or variable cell indicated by Y is a multi-referenced object, in other words, there is one or more RC cells in the reference path, the reference counter in the RC cell that appeared first is incremented. When the data object or variable cell indicated by Y is a single-referenced object, a new RC cell should be inserted, which is pointed by REF● from body goals q and r. When the object is a bound variable cell, namely a value cell, there is only one reference path to the object. Therefore, an RC cell can be inserted without using exclusive access even in a parallel processor system with shared memory. However, if the object is an unbound variable cell, there may be another reference path to the variable cell, as shown in Figure 3(a). In such cases, an RC cell will be inserted by allocating a new RC variable cell, followed by unification, as shown in Figure 4.

3.4 Dereferencing

Dereferencing preceding a unification does not consume reference paths to data objects. However, if there are two or more RC cells in the reference path, as in Figure 5, the pointer can be reconnected directly to the farther RC cell.

3.5 Instantiation of variables

In KL1 goal reduction, a built-in unifier, =, in a clause body unifies[9] two data objects (or variables). One reference path to each data object is consumed by this unification. When one or both data objects are unbound variables, one unbound variable is given some concrete value (atomic or structured) or bound to another variable by connecting them with a reference chain. In this case, a reference path from the instantiated variable to another

[9]This is called *active unification*.

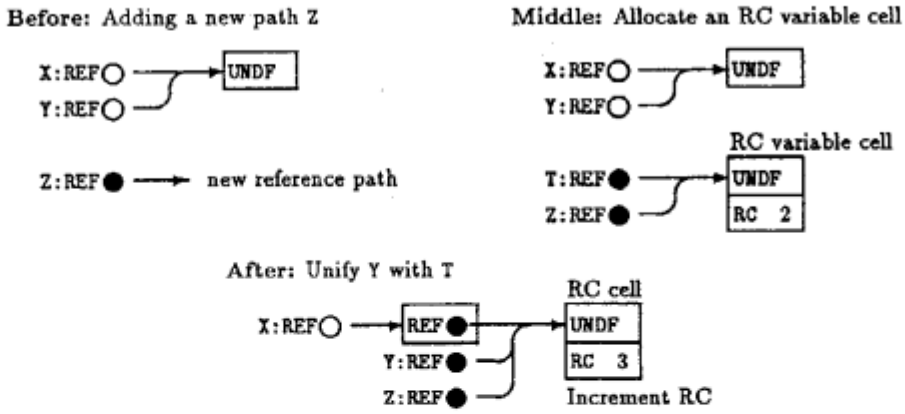

Figure 4: Lazy allocation of an RC cell to an unbound variable

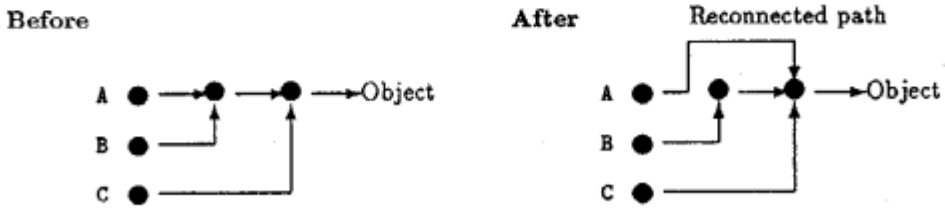

Figure 5: Reconnecting a reference path in dereferencing

variable is added. Therefore, only one reference path to the instantiated variable is consumed, keeping the total number of reference paths to another variable unchanged.

The reference count update rules in unification between X and Y are as follows. Here, X is an unbound variable to be instantiated, and Y is a data object including an unbound variable.

a: When both X and Y are single-reference objects,

bind X with the pointer to Y which is already dereferenced.

b: When Y is a multi-referenced object,

bind X with the pointer to the RC cell which appears first in the reference path to Y.

c: When X is a multi-reference object,

if X is an unbound variable as shown in Figure 7(a), change the unbound variable cell to an RC variable cell as shown in Figure 7(b) first, then decrement the reference count of the RC cell, then bind X with the pointer to Y.

3.6 Retrieval of structure elements

Retrieval of structure elements is required on unification with an explicit data structure in the head or guard of a clause, as in:

```
p([X|Y]) :- true | p(X), q(Y).
```

By this unification, one reference path to the list structure is consumed, and one reference path each to structure elements X and Y is created. When the structure has only one reference path, i.e. when it is a single-referenced structure, the whole structure can be reclaimed and the reference paths from the body of the structure to its

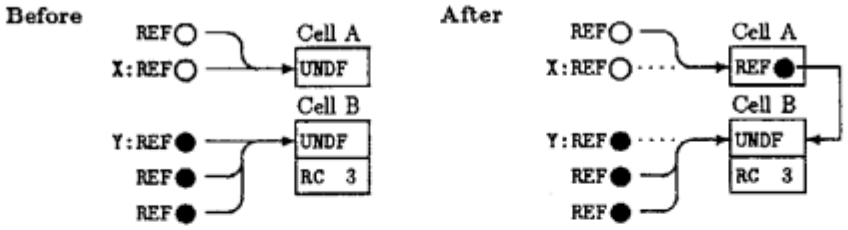

Figure 6: Unification between two unbound variables

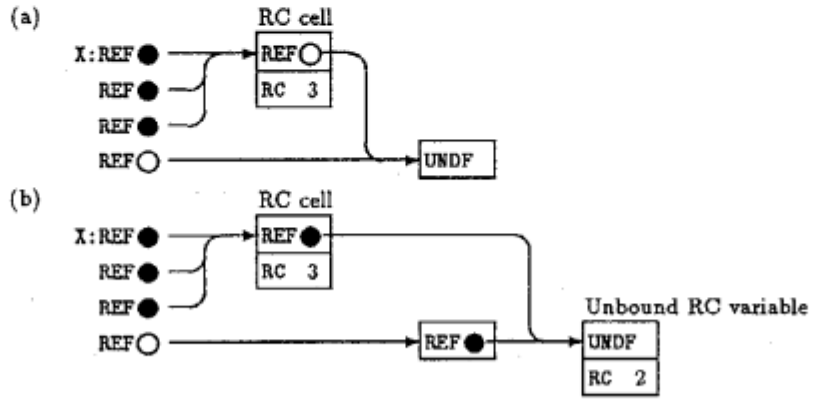

Figure 7: Unbound variable in a multiple reference path

elements also disappear, so that the total number of reference paths to the structure elements is kept unchanged. However, when the structure is a multi-referenced structure, the reference path from the body of the structure to its element remains unchanged, so that the number of reference paths to the structure element is incremented by one. In this case, if the structure element is pointed only by the structure, an RC cell should be inserted. Figure 8 shows an example. Here, when the cdr of list cell b to Y is retrieved, RC cell d is inserted, so that pointers from list cell b and from Y meet at the RC cell.

4 Abstract Machine and Instruction Set

The KL1 abstract machine and its instruction set proposed in [11] are modified to implement LRC real-time garbage collection. This section describes major differences from the original in [11].

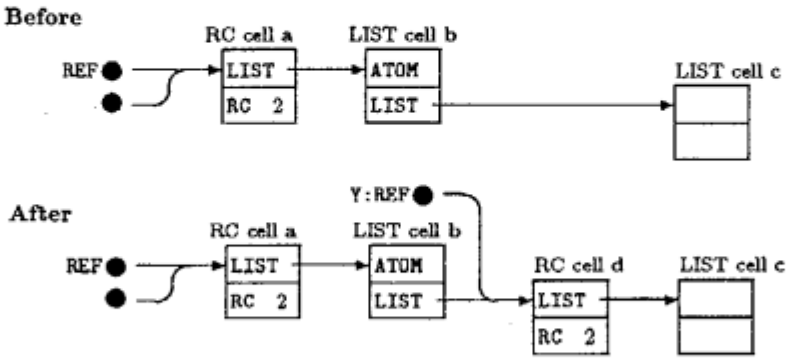

Figure 8: Insertion of an RC cell in retrieving a structure element

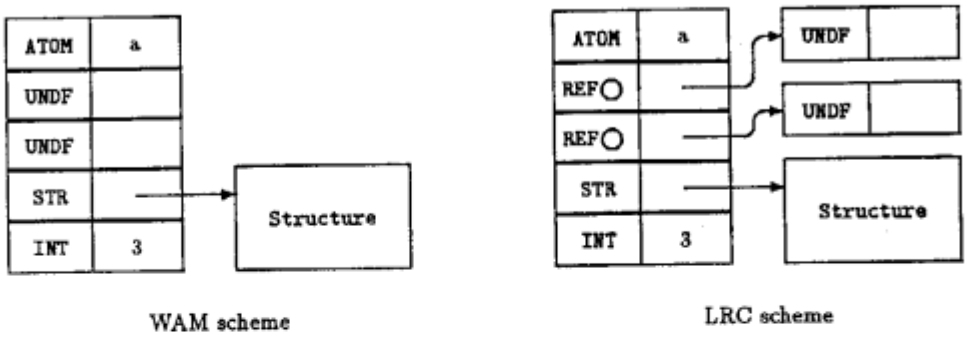

Figure 9: Variables as structure elements

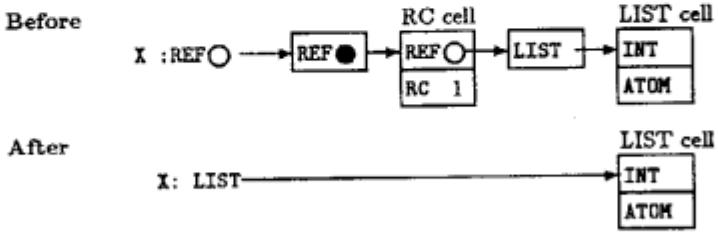

Figure 10: Storage reclamation in dereference

4.1 Abstract machine

First, multiple reference pointer tag REF● is introduced as a new type tag in internal data representation. Next, the free memory area for variable cells or for structures is maintained as a free list to allocate and to reclaim dynamically. Then, unbound variables as structure elements are placed out of the structure body which are pointed by the reference pointer from the body, as shown in Figure 9. By this scheme, the body of a structure and its elements can be reclaimed independently.

4.2 Dereference

When variable values are required in unification, reference pointers between variable cells are dereferenced until the final result such as atomic values, structure pointers or a pointer to an unbound variable cell is reached. Because KL1 has no backtracking feature, the dereferenced result can be stored back to the place where the pointer originally was (on registers or in data structure elements).

As the deferenced result should be stored back, the intermediate variable cells may be reclaimed. Cells that become reclaimable during dereference are instantiated variable cells pointed by single reference pointers and RC cells whose reference counter value is one.

A dereference algorithm follows.

Dereference algorithm

In what follows, two registers are used.

A : An argument register which has a reference path to an object.

S : The structure register which will have a pointer to a structure body, if the dereferenced object is a structure.

Step 1: If A is not an *indirect pointer* (not REF● nor REF○), → **end of dereference.**

Step 2: Dereference the path from A until an instantiated value, a pointer of REF○ to an unbound variable cell, or an RC cell (REF●) appears in A, reclaiming cells pointed by the indirect pointer cell of REF○ ; → **step 3.**

Step 3: Do the following according to the type of dereferenced result. If A is:

an instantiated value ⇒ put A in S, if A is a pointer to a structure; → **end of dereference.**

a pointer of REF ○ to an unbound variable cell ⇒ **end of dereference.**

a pointer (REF ●) to an RC cell ⇒ put the value pointed by A in S; → **step 4.**

Step 4: Do the following according to the type of S. If S is:

either an instantiated value or an unbound RC variable ⇒ **end of dereference.**

an indirect pointer of REF ○ ⇒ store the value pointed by S back in the RC cell pointed by A, reclaim the cell pointed by S, then put the value pointed by A in S; → **step 4.**

an indirect pointer of REF ● ⇒ increment the reference count in the RC cell pointed by S, then decrement the reference count in the RC cell pointed by A; → **step 5.**

Step 5: If the reference count of the RC cell pointed by A is zero, reclaim the RC cell; → **step 6.**

Step 6: Put S in A, then obtain the value pointed by S as a new value of S; → **step 4.**

4.3 Instruction set

The KL1 abstract machine instruction is similar to that of WAM [20]. The major differences are:

- Passive unification instructions will be suspended when instantiation of variables is required to accomplish the unification.
- The guard part is compiled so that argument registers are never destroyed before commitment.
- Instructions are arranged so that reference paths to data objects can be maintained correctly.

To implement LRC incremental garbage collection, the unification instructions should be slightly modified, and several new instructions introduced. The new instructions are for maintaining multiple references and for reclaiming garbage cells.

4.3.1 Instructions to create new multi-referenced variables

```
set_rc_variable         Xi, Gj, rc
put_rc_variable         Xi, Aj, rc
write_rc_variable       Xi,     rc
```

These three instructions allocate new unbound RC variable cells pointed by REF ● , as shown in Figure 3(b). The reference count is initiated as the third argument, `rc`, indicates. Here, the initiated reference count can be determined in compiling time.

4.3.2 Adding reference in clause body

```
add_reference  Xi, rc
```

This instruction is used to pass a variable in a clause head to multiple body goals as described in Section 3.3. If there is one or more RC cells in the reference path to `Xi`, this instruction increases the reference count of the RC cell that appears first. However, if `Xi` is a single-referenced variable, this instruction inserts an RC cell to represent multiple references. This instruction can be combined with **`put_value`**, **`set_value`** or **`write_value`**, as:

```
put_value_add_ref    Ai, Xj, rc
set_value_add_ref    Ai, Gj, rc
write_value_add_ref  Gj,     rc
```

4.3.3 Adding reference of structure elements

```
add_ref_struct_element  Ai, position, Aj
```

When an element indicated by `position` of a structure, `Ai`, was retrieved to `Aj`, this instruction maintains addition of a reference to the element, as mentioned in section 3.6. If `Ai` indicates a multi-referenced structure, i.e. `Ai` has a tag, `REF` ● , this instruction adds a reference path to the element. If `Ai` has a tag, `REF` ○ , this instruction has no operation.

4.3.4 Storage reclamation instructions

```
collect_list Ai
collect_vect Ai
```

These instructions maintain consumption of a reference path to structure, `Ai`, when the elements of the structure are retrieved as described in Section 3.6. Then, if the consumed path is the last one to the structure, that is, if the tag of `Ai` is `REF` ○ or if it is `REF` ● and the reference counter of the RC cell pointed by `Ai` shows one, the memory area is reclaimed. If the structure `Ai` is a multi-referenced object, the reference counter in the RC cell pointed by `Ai` is decremented by one.

```
collect_value Ai
```

This instruction recursively consumes a reference path to the data object, `Ai`, when a general unification or a unification with a void variable occurs in the guard of a clause [11].

4.3.5 Compiling examples

Figure 11 shows compiling examples for LRC garbage collection. Here, the three instructions in Figure 11,

```
add_ref_struct_element A1, car, X3
add_ref_struct_element A1, cdr, X4
collect_list           A1
```

can be replaced by the following one instruction.

```
retrieve_list_elements A1, X3, X4
```

This instruction merge may have considerable efficiency gain because the operations of the three instructions are determined by the same condition, i.e. whether the list cell is a multi-referenced object or not.

5 Garbage Collection in PIM

5.1 Requirement for garbage collection method

A parallel inference machine *PIM* consisting of about 100 processing elements is now being developed at ICOT [8]. The target processor performance is 200 to 500 KRPS[10] for KL1, so that 10 to 20 MRPS is expected as the total performance for actual applications. *PIM* has a hierarchical structure as shown in Figure 12. Each cluster consists of eight or more processor elements (PEs) which share one address space and communicate through shared memory (SM) over a common bus. The clusters are connected by a switching network.

Each PE in the *PIM* has coherent cache memory [1,13], which increases the efficiency of local execution. In addition, exclusive memory access can be obtained at small cost by using the cache block status of coherent cache memory [13]. Parallel processors with shared memory and local coherent cache memory like PIM clusters aim to decrease processor-memory communication relying on locality of memory references. In a PIM cluster, the

[10]RPS: KL1 goal Reduction Per Second

```
p([X|_], X) :- true | q(X,Y), r(Y,[Y|X]).

p/2:  wait_list  A1                          % Unify the first arguments with list.
      read_variable X3                       % Read the car of list.
      read_variable X4                       % Read the cdr of list.
      wait_value X3, A2                      % Unify the 2nd arg with car of list.
                                             - ( commit ) -
      add_ref_struct_element A1, car, X3     % Add the reference count for car
      add_ref_struct_element A1, cdr, X4     % Add the reference count for cdr
      collect_list  A1                       % Consume a path, reclaim if possible.
      collect_value X4                       % Consume a path, reclaim if possible.
      collect_value A2                       % Consume a path, reclaim if possible.
      create_goal r/2                        % Create a goal record.
      set_rc_variable X2, G1, 3              % Create a new RC variable cell.
      set_list G2                            % Create a new list cell.
      write_value X2
      write_value_add_ref X3, 1              % Add the reference count, then write in cdr.
      enqueue_goal r/2                       % Enqueue the goal record.
      put_value  X3, A1                      % Rearrange argument registers.
      execute    q/2                         % Execute goal q/2.
```

Figure 11: Compiling examples

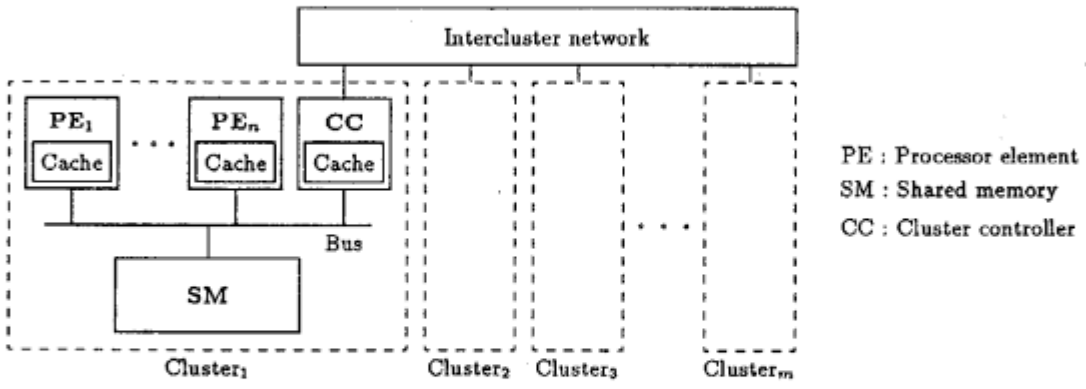

Figure 12: *PIM* overview

memory reference locality is enhanced by an appropriate scheduling and load balancing scheme [14]. Garbage collections in such parallel processors should be as efficient as usual parallel execution. That is, garbage collection should be done in parallel, and memory references during garbage collection should have good locality.

5.2 LRC garbage collection in the PIM

In the implementation of LRC garbage collection on a PIM cluster, data objects must be accessed exclusively to update a reference count of more than one. However, exclusive access can be done at low cost using the hardware lock mechanism. In addition, only one lock at a time is required in the abstract machine instruction set for LRC. As a result, reference information maintenance and garbage cell reclamation in LRC require only memory operations with high locality. Therefore, these operations make best use of cache memories in the PIM.

Finding out the reference path information by LRC enables several interesting techniques in KL1 implementation, such as destructive array update and efficient stream merge, that are also available with the MRB method [2].

5.3 Comparison with other garbage collection schemes

Marking or moving garbage collection schemes must access all data objects, so that garbage collection requires enormous data communication in parallel computer systems. In other words, the garbage collection performance is restricted by the band-width between processors and shared memory because hardware mechanisms, such as a local coherent cache, do not work effectively. Another problem in parallel processor systems with a network is that if one network node stops execution and starts garbage collection, other nodes can not communicate with the garbage collecting node. As a result, the garbage collecting node disturbs all the other nodes.

The most notable difference from the MRB is the storage reclamation ability. Using the MRB method, one goal reduction usually generates about one word garbage on average, though it depends on the characteristics of the source program [12]. Therefore, other non-incremental garbage collection, such as copying garbage collection must be used with MRB garbage collection. In LRC, it is not necessary to use other garbage collection, except for loop structures and fragmentation of memory area. The overhead of LRC is almost comparable to that of the MRB method for single-referenced data objects. However, the cost of changing a single reference to multiple references is high.

6 Conclusion

The LRC method, proposed in this article, has the following features. LRC incrementally maintains multiple reference information using a pointer tag and an RC cell, instead of hash tables. The storage reclamation ability is high enough to solve big application programs without using other garbage collection mechinism. The overhead for single reference data objects is as small as in the MRB method. The additional storage needed for LRC is only for RC cells. LRC can be implemented on parallel processors with shared memory like a PIM cluster efficiently. A detailed evaluation of LRC will be done in the near future.

Acknowledgement

I wish to thank the research members of the PIM and multi-PSI project, especially Mr. K. Nakajima, and Mr. S. Miyauchi for their useful comments. I also wish to thank to ICOT Director, Dr. K. Fuchi, and the chief of the fourth research section, Dr. S. Uchida, for their valuable suggestions and guidance.

References

[1] P. Bitar and A. M. Despain. Multiprocessor cache synchronization. In *Proc. of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[2] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 276–293, 1987. (also in ICOT Technical Report, TR-248).

[3] D.W. Clark and C.C Green. An Empirical Study of List Structure in Lisp. *Commun. ACM*, 20(2):78–86, Febr. 1977.

[4] K. Clark and S. Gregory. Notes on Systems Programming in PARLOG. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 299–306, Tokyo, 1984.

[5] K. Clark and S. Gregory. *PARLOG: Parallel Programming in Logic*. Research Report DOC 84/4, Dept. of Computing, Imperial College, 1984.

[6] J. Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13(3):341–367, Sept. 1983.

[7] L.P. Deutsch and D.G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *CACM*, 19(9):522–526, Sept. 1976.

[8] A. Goto. Parallel Inference Machine Research in FGCS Project. In *US-Japan AI Symposium 87*, pages 21–36, Nov. 1987.

[9] A. Goto and S. Uchida. *Toward a High Performance Parallel Inference Machine –The Intermediate Stage Plan of PIM–*. TR 201, ICOT, 1986. (also in Future Parallel Computers, LNCS 272, 299–320, Springer-Verlag).

[10] Y. Hibino. Garbage Collection and its Hardware. *IPSJ*, 23(8):730–741, Aug. 1982. (in Japanese).

[11] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 468–477, 1987. (also in ICOT Technical Report TR-246).

[12] Y. Kimura, K. Nishida, S. Miyauchi, and T. Chikayama. Realtime GC by Multiple Reference Bit in KL1. In *Proceedings of the Data Flow Workshop 1987*, pages 215–222, Oct. 1987. (in Japanese).

[13] A. Matsumoto et al. *Locally Parallel Cache Designed Based on KL1 Memory Access Characterestics*. TR 327, ICOT, 1987. (also submitted for ISCA 1988).

[14] M. Sato, A. Goto, et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 338–355, 1987.

[15] E.Y. Shapiro. *A subset of Concurrent Prolog and Its Interpreter*. TR 003, ICOT, 1983.

[16] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, 1986.

[17] K. Ueda. *Guarded Horn Clauses: A Parallel Logic Programming Language with the concept of a Guard*. TR 208, ICOT, 1986. (also to appear in Programming of Future Generation Computers, North-Holland, Amsterdam, 1987.).

[18] K. Ueda. *Introduction to Guarded Horn Clauses*. TR 209, ICOT, 1986.

[19] A.H. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

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