TR-904

Reference Loop Management in a Distributed KLIC Implementation

by
K. Rokusawa, T. Chikayama,
T. Fujise & A. Nakase

January, 1995

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~ 5

Institute for New Generation Computer Technology

Reference Loop Management in a Distributed KLIC Implementation

Kazuaki Rokusawa Takashi Chikayama Tetsuro Fujise Akihiko Nakase
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN
{rokusawa, chikayama, fujise, nakase}@icot.or.jp

Abstract

This paper describes the management of interprocessor reference loops in a distributed KLIC implementation. The reference on a reference loop does not have dereferenced results, and the simple forwarding of a read request may not terminate. In a distributed KLIC implementation, reference loops can be created by unification, and remote value fetch operations can create loops as well. A value fetching scheme which can cope with reference loops created by unification, and a collection scheme which can reclaim loops created by remote value fetch are presented.

1 Introduction

This paper describes the management of interprocessor reference loops in a distributed KLIC implementation [1].

A reference loop is a closed chain of references. The reference on a reference loop does not have dereferenced results, and the simple forwarding of a read request message along the loop may not terminate. Although several collection schemes which can reclaim cyclic structures have been proposed, they are costly and operations are complicated [2].

In previous implementations of KL1 [3] on the Multi-PSI and PIMs [4, 5], much work has been done to avoid loop creation by unification. Some attribute is attached to each variable and external reference, and the possibility of loop creation is examined using them before binding [6]. The unifier which performs unification involving an external reference is thus different from the one for intra-processor processing. Although cyclic structures cannot be reclaimed, no problems caused by cyclic structures have been reported.

In a distributed KLIC implementation, exactly the same unifier as the one for the sequential core [7] is used to perform unification involving an external reference, because the implementation is designed to provide a portable system. Since the unifier allows binding of a variable to any reference pointer, loops can be created by unification. In addition, remote value fetch operations can create loops as well. To overcome the problems above, new schemes for value fetching and loop collection were invented. The value fetching scheme can cope with reference loops, and the collection scheme can reclaim reference loops created by remote value fetch operations.

This paper is organized as follows. Section 2 defines external references. Creation of interprocessor reference loops is described in section 3. Solutions to cope with reference loops are presented in section 4.

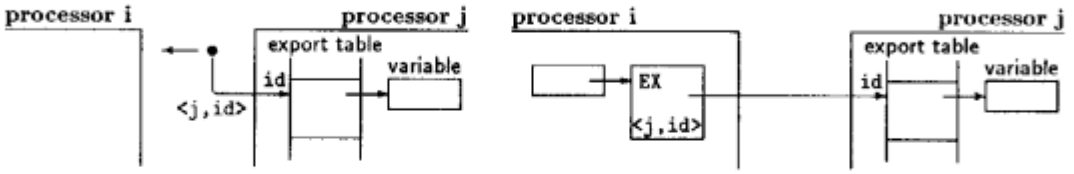


Figure 1: Generation of an external reference

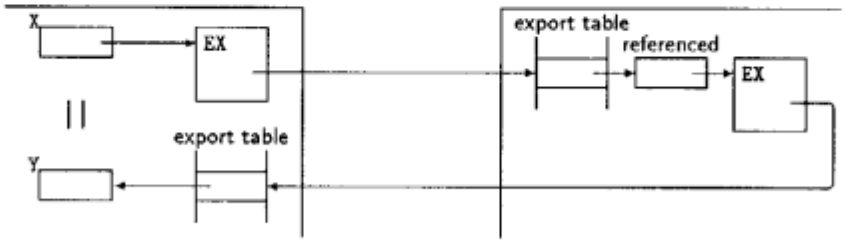


Figure 2: Creation of reference loops by unification

2 External References

When a message is sent to another processor and the message contains references to variables, references across processors consequently appear; these are *external references* (figure 1).

To make it possible to perform local garbage collection independently, each processor maintains an *export table* to register all locations of cells which are referenced from outside. The entries in the export table are ones of roots of local garbage collection. An external reference is represented by a pair $< proc, ent >$, called *exref ID*, where *proc* is the processor number in which the referenced cell resides, and *ent* is the entry number of the export table. When externally referenced cells are moved as a result of local garbage collections, the references from the export table entries are updated to reflect the moves, while the exref IDs are not affected.

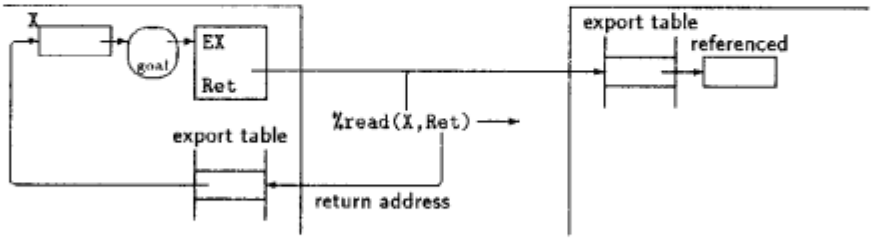
3 Creation of Interprocessor Reference Loops

In a distributed KLIC implementation, interprocessor reference loops can be created by unification and remote value fetch. This section presents the creation of loops in both cases.

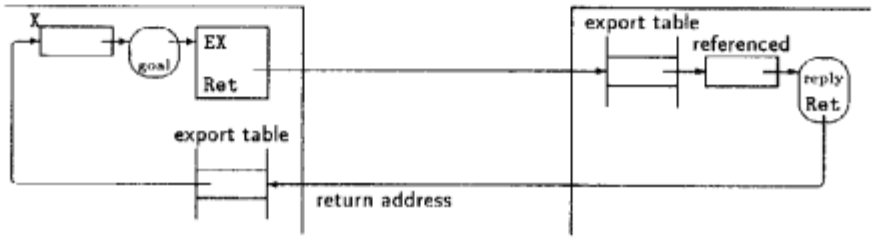
Loops Created by Unification

In a distributed KLIC implementation, loops can be created by unification, because a unification scheme of binding a variable to any reference pointer is employed. Even if a variable may be referenced from outside, the unification between an external reference and the variable is done by simply binding. Therefore, interprocessor reference loops can be created by unification (figure 2).

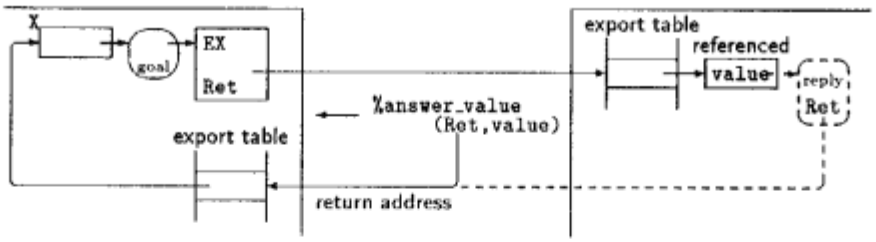
(1) Before fetching



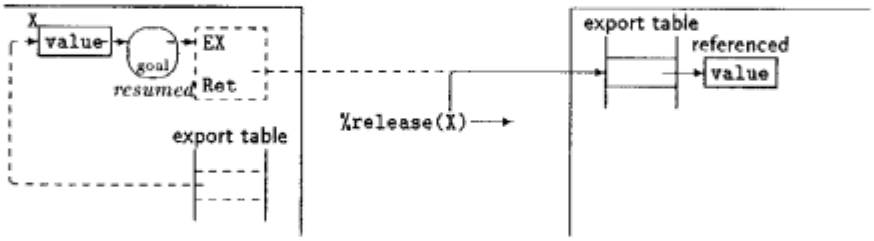
(2) Generating a return address and sending a `%read` message.



(3) Generating and hooking a *reply record*.

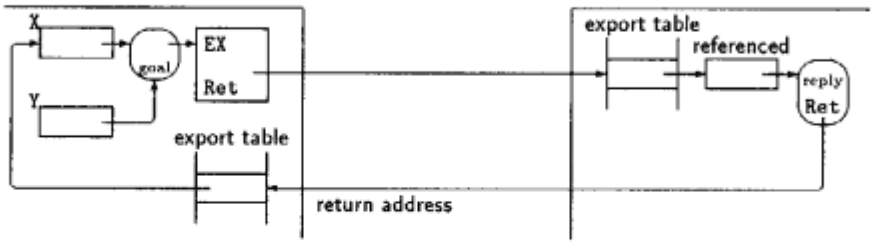


(4) Sending back an `%answer_value` message.

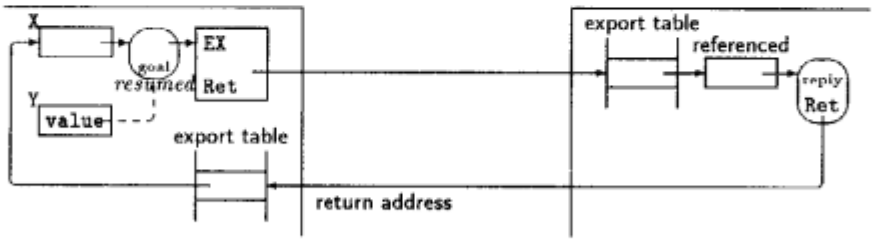


(5) Release of an external reference

Figure 3: Fetching a value from remote memory



(1) A goal is waiting for the value of either X or Y.



(2) The goal can be resumed without receiving an %answer_value message.

Figure 4: Creation of reference loops by remote value fetch

Loops Created by Remote Value Fetch

When a goal reduction requires the value referenced by an external reference X, the following read request message is sent to the processor referenced by X.

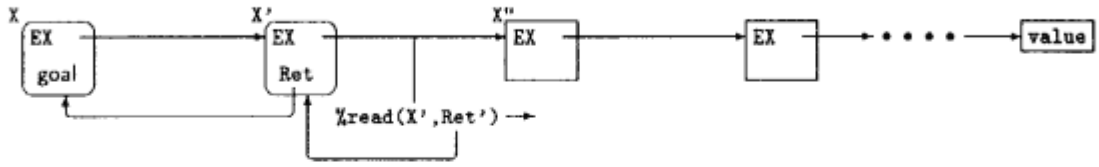
```
%read(X,Ret)
```

Ret is a newly generated external reference to X, which indicates the *return address* for the response of the %read message. The goal waiting for the response is hooked to the external reference (figure 3 (1)(2)).

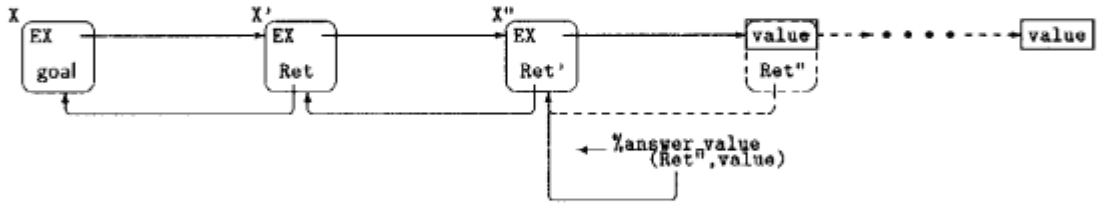
When a %read message arrives at the referenced processor and the referenced cell is a variable, the return of the value is suspended. A record called a *reply record* which memorizes the return address is generated and hooked to the variable. When the referenced cell is an external reference with reply records, the same operations are performed. The original external reference and the corresponding reply record form an **interprocessor reference loop**; the external reference points to the record, and the record points to the external reference as well (figure 3 (3)).

When a goal waiting for the response of the %read message resides at the external reference, the loop is *active*. It is expected that the referenced cell is instantiated with a value and the loop is consequently reclaimed; The reply record associated with the cell is released and an %answer_value message which carries the value is sent to the return address memorized; When the %answer_value message arrives, suspended goals are resumed and the original external reference is released (figure 3 (4)(5)).

However, when the goal is waiting for the value of one of a set of variables and external references, the goal can be resumed without receiving the response (figure 4). In addition, the instantiation may not be performed, and the loop eventually becomes garbage which cannot be locally collected. This situation is common in executing nondeterministic or speculative computations. Therefore, even correct programs can create garbage loops.

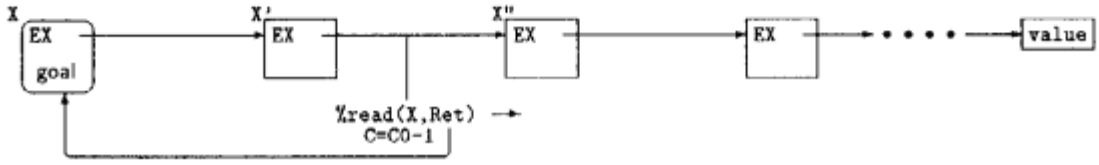


(1) A reply record is hooked on each external reference.

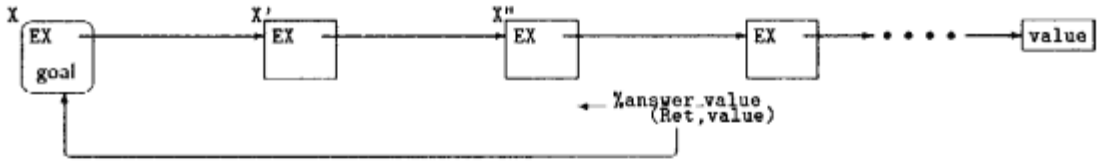


(2) An `%answer_value` message is returned along the same path.

Figure 5: *Hooking Reply Records* Scheme



(1) No reply records are hooked.



(2) An `%answer_value` message is returned directly.

Figure 6: *Using a Counter* Scheme

4 To Cope With Reference Loops

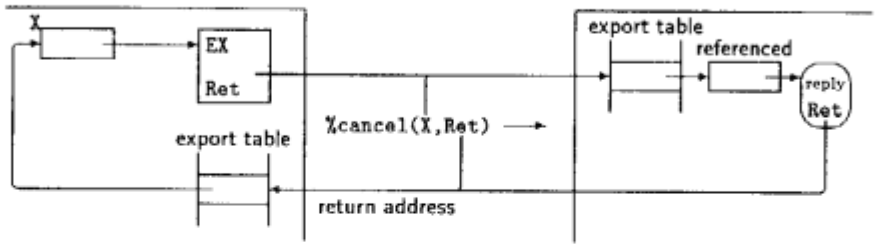
This section describes solutions to cope with reference loops mentioned in the previous section.

Loops Created by Unification

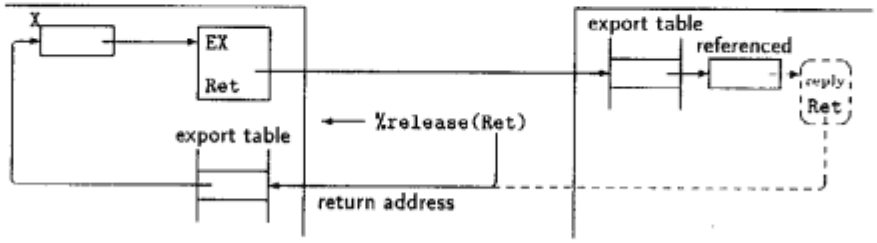
In this subsection, a value fetching scheme which can cope with reference loops is presented.

On receiving a `%read` message, if the referenced cell is an external reference X' without reply records, the `%read` message is forwarded to the processor referenced by X'. The return address is not changed so that an `%answer_value` message can be returned directly. However, simply forwarding a `%read` message may result in a non-terminating operation, since the `%read` message may go into an interprocessor reference loop which consists of only external references with no reply records.

If the value fetching operation is reoriginated on receiving a `%read` message, that is, if a reply record is generated and hooked, and a `%read` message with the return address initialized



(1) Sending a `%cancel` message.



(2) Release of the reply record.

Figure 7: Reclamation of reference loops created by remote value fetch

is sent, termination of forwarding is guaranteed. Even when a `%read` message goes into a reference loop, it is guaranteed that the message reaches such a processor where the referenced cell is an external reference with reply records. However, this *hooking reply records* scheme has the following serious disadvantages (figure 5);

- It is necessary to generate useless reply records even when a reference loop does not exist;
- An `%answer_value` message cannot be returned directly; it is returned along the same path as a `%read` message passed through.

To cope with reference loops, we have introduced a counter for a `%read` message which indicates the maximum number of forwardings. The value fetching scheme using the counter is called a *using a counter* scheme.

An original `%read` message is sent with the counter initialized. When a referenced cell is an external reference with no reply records, the value of the counter is checked. If the value is more than one, the `%read` message is forwarded with the counter decremented. If the value is one, the value fetching operation is originated; a reply record is generated and hooked, and a `%read` message is sent with both the counter and the return address re-initialized (figure 6).

Setting the initial value of the counter at the number of processors, it is expected that no useless reply records are generated and an `%answer_value` message is always returned directly to the original external reference.

Loops Created by Remote Value Fetch

This subsection describes a scheme which can reclaim reference loops created by remote value fetch operations.

If an external reference that has already sent a read request has neither associated goals nor reply records waiting for responses, the read request can be cancelled. Cancelling a read request leads to the release of the reply record which memorizes the return address of the cancelled read request, and the loop can eventually be reclaimed.

Reclamation of reference loops is performed as a part of local garbage collection. Details of the operation are as follows. On each external reference X that has already sent a read request but has neither associated goals nor reply records waiting for responses, the following cancel request message is sent to the processor referenced by X.

```
%cancel(X,Ret)
```

`Ret` is the return address which indicates the read request to be cancelled.

When the `%cancel` message arrives at a processor, if the referenced cell has a reply record with the same return address, the record is released and a `%release` message is sent back to the return address (figure 7). If the cell has a concrete value, nothing is done. This is for the case where an `%answer_value` message has already been sent.

5 Summary

In a distributed KLIC implementation, interprocessor reference loops can be created by both unification and remote value fetch operations. To cope with reference loops, we invented the following schemes.

- A remote value fetch scheme which ensures termination of forwarding a read request.
- A collection scheme which can reclaim reference loops created by remote value fetch operations.

The remote value fetch scheme is based on counting the number of forwardings. Collection of reference loops is performed by cancellation of read requests.

References

[1] K. Rokusawa, A. Nakase, and T. Chikayama, "Distributed Memory Implementation of KLIC," *Proc. Workshop on Parallel Logic Programming and its Programming Environments*, Technical Report CIS-TR-94-04, University of Oregon, pp.151–162, March, 1994. Also ICOT Technical Report, to appear.

[2] D. R. Brownbridge, "Cyclic Reference Counting for Combinator Machines," *Proc. Functional Programming Languages and Computer Architecture*, LNCS 201, pp.273–288, 1985.

[3] K. Ueda and T. Chikayama, "Design of the Kernel Language for the Parallel Inference Machine," *The Computer Journal*, Vol.33, No.6, pp.494–500, 1990.

[4] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama, "Distributed Implementation of KL1 on the Multi-PSI/V2," *Proc. International Conference on Logic Programming*, pp.436–451, 1989.

[5] K. Hirata, R. Yamamoto, A. Imai, H. Kawai, K. Hirano, T. Takagi, K. Taki, A. Nakase, and K. Rokusawa, "Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1," *Proc. International Conference on Fifth Generation Computer Systems,* pp.436–459, 1992.

[6] N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura, "A New External Reference Management and Distributed Unification for KL1," *New Generation Computing,* Ohmsha Ltd., pp.159–177, 1990.

[7] T. Chikayama, T. Fujise, and D. Sekita, "A Portable and Efficient Implementation of KL1," *Proc. International Symposium on Programming Language Implementation and Logic Programming,* LNCS 844, pp.25–39, 1994.