PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

A NEW EXTERNAL REFERENCE MANAGEMENT AND DISTRIBUTED UNIFICATION FOR KL1

Nobuyuki Ichiyoshi, Kazuaki Rokusawa, Katsuto Nakajima and Yu Inamura

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

ABSTRACT

This paper describes a new external reference management scheme for KL1, a committed choice logic programming language based on GHC. The significance of the new scheme is that it realizes incremental inter-processor garbage collection. Previous distributed implementations of committed choice languages had not seriously address inter-processor garbage collection.

Incremental inter-processor garbage collection is realized by the Weighted Export Counting (WEC). It is a first attempt to use the weighted reference counting technique in logic programming language implementation, and is also new in that it has introduced export and import tables for making independent local garbage collection possible and reducing the number of inter-processor read requests.

The problems with exhaustion of reference counts and indirect exportation are discussed. Since the binding order rule adopted in our previous implementation for avoiding creation of reference loops is insufficient in the presence of indirect exportation, a new binding order rule is introduced. We prove that avoidance of reference loops is guaranteed and also prove that the unification procedure always terminates for non-circular structures.

1 INTRODUCTION

GHC [Ueda 86], like Concurrent Prolog [Shapiro 83] and Parlog [Clark 86], is a logic programming language designed to exploit AND-parallelism in logic programs. The reason why we pursue AND-parallelism in favor of OR-parallelism or restricted AND-parallelism (RAP) is that AND-parallelism captures the notion of interacting processes. Interacting processes arise naturally in the real world: problem solving by multiple agents, or open systems that interact with the outside world. KL1 is based on GHC, but is extended with metaprogramming and load distribution capabilities, to make it a suitable language for writing operating systems and for conducting research in load balancing.

We are developing the Parallel Inference Machine (PIM) [Goto 87] and the Multi-PSI [Taki 86] to run large KL1 programs for AI and other applications. The PIM is made up of loosely-coupled systems (called *clusters*) consisting of multiprocessors sharing local memory. The Multi-PSI is made up of up to 64 loosely-coupled processors (CPUs of Personal Sequential Inference Machines (PSIs)) with separate local memory.

The study of KL1 implementation has led us to recognize the importance of garbage collection in AND-parallel language implementations: Garbage collection can take up a significant processing time, thus degrading the overall performance of the system. The major reasons are: (1) an AND-parallel language does not have destructive assignment of variables (as in Fortran), (2) it does not allow stack-based reclamation of control frames (as in Lisp), and (3) it does not have automatic garbage reclamation on backtrack (as in Prolog). We have found out that conventional garbage collection schemes could slash the effective performance of the system by half or more depending on how much memory cells are active. Some solutions have been proposed of late (MRB [Chikayama 87], LRC [Goto 88], Piling GC [Nakajima 88]) for non-distributed models of implementation. Previous distributed implementations of AND-parallel languages [Taylor 87, Ichiyoshi 87, Foster 88a], however, did not address garbage collection issues seriously.

This paper describes a new external reference scheme that has a builtin incremental inter-processor garbage collection mechanism, called the Weighted Export Counting (WEC). It is a generalization of standard reference counting. By assigning weighted reference counts to pointers (references) as well as to referenced data, it has solved the racing problem in a distributed environment. Though the technique has been used in functional language implementations [Bevan 87, Watson 87] on multiprocessors, our external reference management scheme is a first attempt to use the technique for logic programming. It differs from [Bevan 87] and [Watson 87] in that it has introduced the export tables for making independent local garbage collection possible, and the import table for reducing the number of inter-processor read requests. The problem of exhaustion of reference counts is more fully discussed. In particular, the problems with indirect exportation — exportation of imported reference — are pointed out.

Distributed unification is a vital feature in a distributed implementation of logic programming languages. It turns out that, under the new external reference management scheme, the binding order rule in [Ichiyoshi 87] can no longer prevent reference loops to be created, because of the existence of indirect exportation. We propose a new binding order rule to fix this problem, and prove that creation of reference loops is in fact avoided. We also prove that the unification procedure always terminates for non-circular structures.

A strategy for allocating and dividing reference counts is briefly mentioned. Under the strategy, the exhaustions of reference counts are expected to be sufficiently rare so that the extra overhead caused by exhaustions will not affect the overall performance of the external reference mechanism.

2 KL1 LANGUAGE OVERVIEW

In this section, we give a sketch of the KL1 language specification.

KL1, which stands for Kernel Language version 1, is a concurrent logic programming language. It is Flat GHC augmented with metaprogramming and load distribution capabilities. [1] Unlike GHC which is a theoretical language, KL1 is designed as a practical language to write an operating system and application programs to execute on multiprocessors.

A collection of Guarded Horn Clauses makes up a KL1 program. They are of the form:

$$\underbrace{H :- G_1, \ldots, G_m}_{\text{guard}} | \underbrace{B_1, \ldots, B_n.}_{\text{body}} \ (m > 0, \ n > 0)$$

where H, G_i, and B_i are atomic formulas. H is called the *head*, G_i the *guard goals*, B_i the *body goals*. The vertical bar (|) is called the *commitment operator*.

The logical reading of the clauses is the same as GHC [Ueda 86]. KL1 is flat in that only the predefined set of builtin predicates are allowed as guard goals and thus goals cannot nest in the guards.

The metaprogramming capability of KL1 is realized by the *shoen* (pronounced 'sho-en') facility. While goals executed tail-recursively (*processes*) define small-grain threads of control, a shoen defines a larger-grain computational unit. It deals with exception handling and resource management. A shoen is created by a call to the builtin predicate execute/7:

```
execute(Min,Max,Mask,Code,Argv,Control,Report)
```

`Min` and `Max` are minimum and maximum possible priorities allowed in the shoen. `Mask` is a bit pattern for determining which exceptions to handle in this shoen. `Code` and `Argv` specify the initial goal (the predicate code and its arguments) to execute in the shoen. `Control` and `Report` are the control and the report streams. Exceptions that have occurred in the shoen or are delegated from one of the child shoens are reported to the report stream if the logical AND of the exception tag and the exception mask of the shoen is not zero. The control stream is to start, stop or abort the shoen from the outside.

An exception is reported as a message to the report stream, and the monitoring process is to substitute a new goal for the goal that has given rise to the exception. An important thing to note is that there is no failure in a shoen. Any kind of failure is treated as an exception.

Currently, load distribution is realized by means of *pragmas* [Shapiro 84] of the form *@processor(Proc)*, attached to body goals as postfixes. A body goal *P@processor(PE)* is *thrown* to processor *PE* when the clause containing the goal is committed to. The semantics of programs with pragmas is the same as that with the pragmas removed. In the future, we plan to implement a dynamic load balancing mechanism.

The priority of execution is specified by a different kind of pragma (*@priority(P)*).

In our KL1 implementation, guard unification between two unbound variables always suspends, even if they are identical. The reason is that such unification is very rarely needed but still needs be taken special care (a new message must be introduced, etc.). [Foster 88a] has a special *ns_read* protocol for variable-variable unification.

3 MACHINE ARCHITECTURE AND DISTRIBUTED IMPLEMENTATION

3.1 Machine architecture

The machine we assume in the paper is a loosely-coupled multiprocessor. More specifically,

1. The machine consists of a finite number of processors identified by serial identification numbers $(0, 1, \ldots)$.
2. The constituent processors have local memory separate from others.

[1] Actually, KL1 is a hierarchical family of languages comprising the abstract machine language KL1-B(base), the concurrent logic language KL-C(core), the pragma extension KL1-P(pragma), and a collection of user languages KL1-U(user). Here we are talking about KL1-C and KL1-P.

3. The processors are interconnected by a network so that a processor can communicate with any processor by message passing.

We assume that inter-processor communication is expensive. A message sending and receiving overhead is assumed to be roughly 100 times that of simple access to the local memory. We also assume that inter-processor communication is rare compared to local memory accesses. These assumptions justify the rather complicated external reference scheme we adopted.

The Multi-PSI is one such multiprocessor. The PIM's clusters correspond to processors in the above model. The network of the Multi-PSI version 2 guarantees that the communication channel between any two processors is first-in-first-out (FIFO). The PIM may not support FIFO communication,

3.2 Distributed implementation

In the distributed implementation of KL1 on a loosely-coupled multiprocessor, each processing element (PE) executes the reduction cycle independently. That is, each PE has its own scheduling queue of goals and, it picks up a goal of the highest priority and tries to reduce it into body goals. The reduction may fail (and cause *failure exception*), suspend, or succeed. In the last case, the body goals are put to the scheduling queue or *thrown* out to other PEs according to the pragmas.

Throwing of a goal is done by means of the *throw* message in which are encoded the code of the predicate of the goal, the arguments of the goal, the shoen to which the goal belongs plus other bookkeeping information. The encoding and decoding of arguments (or any KL1 data) are respectively called *exportation* and *importation*.

4 EXTERNAL REFERENCES

We assume without loss of generality that the data types in KL1 are atomic data and vectors (represented by a tagged pointer to n consecutive cells). A cell can either hold a concrete value as above, point to another cell (in case of a `REF` cell), represent an unbound variable (`UNDEF` cell) [2], or be either one of the external reference cells (`EXREF` and `EXVAL` cells) defined below.

4.1 Representation of external references

In the distributed implementation, a reference can be *external* as well as *internal*. An external reference is a reference to a non-local data and is represented by the pair $< pe, ent >$, where pe is the PE number in which the referenced data resides, and ent is the unique identification number of location of the data in that PE. (When we talk of a data, it means a physical representation of a logical term.)

We did not choose to take the memory location as the unique identification number. This is because that would make local garbage collection (garbage collection within one PE) very difficult. If the locations of data move as the result of the local garbage collection, it must be announced to all PEs that may have the reference to the data. Instead, each PE maintains an *export table* to register all locations that are referenced from outside. Each externally referenced cell is pointed to by an entry in the table, and the entry number is used as the unique identification number. When the externally referenced cells are moved as the result of a local garbage collection, the pointers from the export table entries are updated to reflect the moves.

A hash table is attached to the export table so that in case a cell is exported more than once the same export table entry may be retrieved from the cell address and used in the second and later exportation.

Also, each PE maintains an *import table* to register all imported external references. All references in a PE to the same external reference are represented by internal references to the same *external reference cell*. The external reference cell and the import table entry point to each other. (The reason for separating the cell and the entry is explained in Section 4.4.) There is a hashing mechanism for retrieving an import table entry from an external reference, so that even if a PE imports the same external reference more than once, only one external reference cell is allocated. Export and import tables are shown in Fig. 1.

The introduction of export and import tables helps reduce the number of inter-PE read requests. Suppose PE_i exports the same data X twice to PE_j as an argument to goals P and Q. Only one export table entry and one import table entry are allocated by the hashing mechanism. Even if both P and Q attempt to read X', only one read request message is sent to $PE_i$by the first read request. The second read request does not result in message sending, since the fact that a read message has already been sent is remembered in the export table entry. This mechanism also prevents PEs from making duplicate local copies of the same external data.

An external reference cell is either an `EXREF` cell or an `EXVAL` cell. The data referenced by an `EXVAL` cell is known to be a concrete value. In the rest of the paper, where it does not matter whether the referenced data is a concrete value or not, we refer to an external reference cell as an `EX` cell. For an external reference E, we denote the `EX` cell by $from(E)$, the referenced data (after internal dereference) by $to(E)$. Also for any (phys-

[2]There are several types of unbound variables: hook variables, multiple hook variables, etc. in the actual KL1 implementation. The distinction is mainly for optimizations and is not essential in the discussions in this paper.

ical) data X, the PE in which it resides is denoted by $processor(X)$.

4.2 Exportation of data

In general, a data is created in one PE and then *exported* to other PEs by messages. The PEs that receive the messages *import* the data and, as the result, have the internal representations of the same logical term as the original one.

Exportation of a term is done by *encoding* it in an inter-PE message and sending the message to the target PE. There are three ways of encoding:

encoding by value (in case of a concrete value) To encode the term into a byte sequence representing the value.

encoding by location (in case of non-atomic term) To encode the term by first registering the location of the term in the export table to obtain the entry *ent* and encoding it as $< pe, ent >$, where *pe* is the PE number of the exporting PE.

encoding by reference (in case of an EX cell with external reference pair $< pe, ent >$) To encode the cell as $< pe, ent >$.

A vector can be either encoded by value (by the sequence of vector tag, vector length, and the elements that are encoded recursively, in either one of the three ways), or encoded by location. Since vectors can be nested, the encoding procedure can also nest. It is desirable to predetermine a certain fixed level that the encoding process can nest, because the entire structure is not always needed in the importing PE and, more importantly, because there can be circular structures. When an encoding algorithm stops at level n, it is called a *level n encoding*. The substructures at that level are encoded by location (except for atomic data). Encoding by location can be considered to be level 0 encoding.

An EX cell can be either encoded by reference or by location. In normal situations, the encoding by reference is used. The latter is called an *indirect exportation*. It corresponds to the insertion of indirection cells in [Bevan 87] and [Watson 87]. Indirect exportation was not present in the previous distributed implementations [Ichiyoshi 87, Foster 88a].

Encoding can also be categorized by purpose as follows:

encoding to pass To encode a term so that the PE that imports it may have an internal representation of the same logical term.

encoding to access To encode an external reference to access it, i.e. to read it or to write on it (unify with some term). To encode an external reference to access, it must be encoded by reference. The target PE of this encoding is always the PE that has exported the external reference.

encoding to return value To encode a concrete value in reply to a read request. To encode a term to return value, it must be encoded by value.

The results of encoding a term X in a message are denoted by `pass(X)`, `access(X)` and `value(X)`, respectively, for the three types of encoding.

4.3 Importation of data

When a PE imports an encoded term, it decodes it into an internal representation in the following way. (1) If the term is encoded by value, it is translated into the suitable concrete term. (2) If the term is encoded by location or by reference, there are two cases.

self-importation If the referenced PE is the same as the current PE (importing PE), the export table entry is retrieved from the entry number. The data it points to is the internal representation.

nonself-importation If the referenced PE is not the same as the current PE, the import table is looked up with $< pe, ent >$ as the key. If there is already a corresponding entry, the EX cell it points to is the internal representation. Otherwise, a new entry and a new EX cell are allocated, and the EX cell becomes the internal representation.

Self-importation arises when a PE (say, PE_i) exports a data to another PE (say, PE_j) using the encoding by location, and then PE_j exports it back to PE_i using the encoding by reference.

4.4 Access protocols

In KL1, unification in the guard and in the body are respectively called *passive unification* and *active unification*. The former is a pattern matching without binding any variables, whereas the latter is a pattern matching with possible binding of variables as by-product.

In passive unification, the two terms to be unified are read and compared. To read an EX cell X, a read request is made by sending a *%read* message (shown below) to the referenced PE.

```
%read(access(X),ReturnAddr)
```

`ReturnAddr` is an external reference to the EX cell. [3]

If the referenced cell has a concrete value V, it is returned by an *%answer_value* message:

[3]The *%read* and *%answer_value* messages correspond to the *%read_value* and *%return_value* messages in [Ichiyoshi 87].

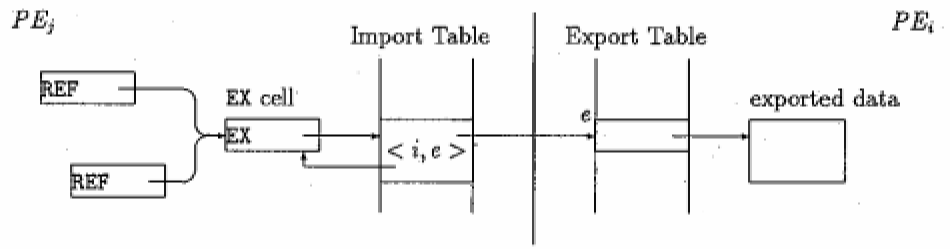

Figure 1: Export Table and Import Table

```
%answer_value(ReturnAddr,value(V))
```

If the referenced cell is an unbound variable, returning of value is suspended. If it is an EX cell, a *%read* message is passed on to the *PE* it references.

When the *%answer_value* message arrives, the EX cell identified by ReturnAddr is overwritten by the value, and the import table entry corresponding to the EX cell can be freed. This is why the cell and the entry are separate.

Remote writing is realized by the unify protocol. Writing a variable that is external to the PE is realized by sending a *%unify* message to the referenced PE. Specifically, to unify an EX cell X with a term Y,

```
%unify(access(X),pass(Y))
```

is sent. It is a request to unify the data referenced by X with a term Y. The PE that receives the above message does the active unification after translating the two terms into internal representations.

5 INTER-PE GARBAGE COLLECTION BY WEIGHTED EXPORT COUNTING (WEC)

In this section, we give a motivation for Weighted Export Counting (WEC) scheme, state its principle, and describe how WEC is maintained at exportation and importation of data. Lastly, the problem of unsplittable WEC is discussed.

5.1 The WEC principle

Since export table entries cannot be freed by a local garbage collection, there must be some inter-PE garbage collection mechanism to free those entries that have become garbage.

One way of realizing inter-PE garbage collection is by a *global garbage collection*. A serious problem with global garbage collection is that it is expected to take a very long time.

Another is an incremental inter-PE garbage collection. One of its merits is that it keeps intact the locality of data access in the program. But, a naive implementation of the standard reference counting scheme, however, does not work correctly in a distributed environment.

Suppose we introduced two messages for incrementing and decrementing reference counts. When a PE discards an external reference, it sends a *%decrement* message to the referenced export table entry. When a PE duplicates an external reference to give it to another PE, it sends an *%increment* message to the entry. The problem here is that before the *%increment* message arrives at the entry, the following sequence of events may take place: The duplicated reference arrives at a PE and is discarded there and the resulting *%decrement* message is received by the entry, causing the entry to be freed. This is a typical racing situation. Note that the FIFO assumption on direct communication between two processors does not say that indirect communication takes more time than direct communication.

Unlike the standard reference counting which assigns reference counts only to referenced data, the Weighted Export Counting (WEC) scheme, assigns reference counts, or weighted export counts (WEC), to references (pointers) as well. More precisely, positive values are assigned to external references (import table entries and references encoded in messages), and non-negative values are assigned to export table entries, so that the following invariant is true for every export table entry E (Fig. 2):

$$(weight\ of\ E) = \sum_{x:\ reference\ to\ E} (weight\ of\ x)$$

It follows from the above equality that the following two are equivalent.

1. The weight of E is zero.
2. There is no reference to E.

This technique of using weighted reference counts has been employed in functional language implementations (WRC in [Bevan 87] and [Watson 87]). The problem peculiar to logic programming language is treated in Section 6. The differences between WRC and our WEC are summarized in Appendix A.

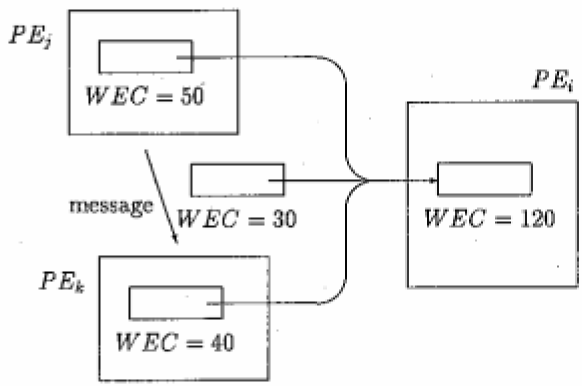

Figure 2: WEC Invariant

In retrospect, the problem with the reference counting GC was that the assumed invariant that the reference count of the export table entry was equal to the number of references was *not* actually an invariant relation, as was shown in the racing example.

5.2 WEC operations

The operations on WEC at exportation are as follows:

encoding by location Add a certain positive weight w to the WEC of the export table entry (if it is newly created, the WEC is initialized to w), and assign w to the encoded result.

encoding by reference Subtract a certain positive weight w from the WEC of the import table entry, and assign w to the encoded result.

There are three kinds of transition of the state of external references, caused by exportation and internal operations of a PE.

duplication The external reference is duplicated: the EX cell is encoded by reference and is retained. The WEC is split in two positive weights.

discard The external reference is discarded: the EX cell and the corresponding import table entry are freed and the associated WEC is returned back to the export table entry by a *%release* message. An EX cell can be freed by the MRB mechanism [Chikayama 87] or by other form of local garbage collection.

transfer The external reference is transferred to another PE: when the number of internal references to the EX cell becomes zero after the encoding, the EX cell and the corresponding import table entry are freed and the associated WEC is given to the encoded result. This situation can be detected if the implementation supports the MRB or other local reference counting scheme.

Example 1 *Here are examples of (1) reference duplication, (2) discard and (3) transfer. We assume that when the goal* a(X) *is executed on PE 12,* X *is the last internal reference to an* EX *cell referencing a data DX in PE 34.*

```
(1) a(X) :- true | b(X), c(X)@processor(56).
(2) a(X) :- true | true.
(3) a(X) :- true | c(X)@processor(78).
```

In (1), the reference to DX is duplicated: one reference is retained in PE 12 and another is exported (by reference) to PE 56. In (2), the reference to DX is discarded. A %release message is sent to PE 34 and the EX *cell together with the import table entry is freed. In (3), the reference to DX is transferred to PE 78. All WEC is encoded into the throw goal message. The* EX *cell together with the import table entry is freed.*

When a PE imports an external reference with encoded WEC of w, the following WEC operation is carried out according to the kind of importation.

self-importation Subtract w from the WEC of the export table entry. If it becomes zero as the result, the entry is freed.

nonself-importation Add w to the WEC of the import table entry (if it is newly created, the WEC is initialized to w).

5.3 Unsplittable WEC and indirect exportation

WEC is implemented as integer on real machines because the invariant must be an exact relation. Since an imported external reference can be duplicated arbitrarily many times, the situation where the associated WEC can no longer be split (i.e. WEC becomes 1) may be reached. There are two ways to cope with this situation.

WEC supply The duplication is suspended and a *%request_WEC* message is sent to the exporting PE. When the message is received, a *%supply_WEC* message carrying a WEC to supply is sent back to the referencing PE. The reference duplication resumes when the *%supply_WEC* message arrives. [4]

indirect exportation The EX cell is not encoded by reference but by location. This involves no suspension of reference duplication, but makes the external reference chain longer. [Bevan 87] and [Watson 87] take this approach.

[4] Actually, a PE may import the same external reference (which always has an assigned WEC) before the *%supply_WEC* message, and that can resume the duplication.

The second method is easier to implement and works fine in the case of encoding to pass, but it cannot be used in the case of encoding to access, since encoding by reference is the only way to access an exported data. If the network has a FIFO property, this problem can be solved as follows.

zero encoding To encode to access an external reference with $WEC = 1$, encode it by reference with $WEC = 0$.

We call such encoding and access *zero encoding*, and *zero access*, respectively. Since the *%release* message that might follow will not overtake the zero access message (FIFO property), the referenced export table entry is guaranteed to exist when the zero access message arrives.

One inconvenience with the introduction of zero access is that reference transfer cannot be done after sending a zero access message. This is because the transferred reference can be discarded and the resulting *%release* message can arrive *before* the zero access message arrives.

Therefore the fact that a zero access message has been sent must be remembered to prevent a reference transfer. The import table entry has a *zero flag* for this purpose. When the zero flag is ON, the external reference must not be transferred but must be indirectly exported. [5]

6 DISTRIBUTED UNIFICATION

6.1 Avoidance of reference loop creation

A *reference loop* is a closed chain of references (internal and/or external). If there were a reference loop, the cells on the loop would not have dereferenced results, and they could not be unified with any concrete value.

In a sequential implementation, creation of reference loops can be avoided by fully dereferencing both reference chains before unifying them. In a distributed implementation, however, two chains cannot always be fully dereferenced at once because the dereferenced results may be two unbound variables in separate PEs. An unrestricted unification algorithm can create reference loops as in the following example.

Example 2 *PE_i has an* EXREF *cell Y' that references an unbound cell Y in PE_j, and PE_j has an* EXREF *cell X' that references an unbound cell X in PE_i. If active unification between X and Y' in PE_i causes X to be bound to Y', and active unification between Y and X' in PE_j causes Y to be bound to X', a reference loop is created. (Fig. 3)*

[5]This is when the WEC is still 1. When the PE imports the same external reference, the WEC increases, and it can be split in two. When a non-zero access message is sent, the zero flag of the import table entry can be reset.

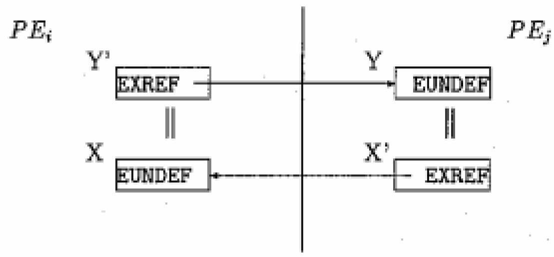

Figure 3: Reference Loop

In [Ichiyoshi 87] and [Foster 88a], the problem is solved by imposing the binding order rule: a binding of an unbound variable to an EXREF cell by active unification is permitted only when the current PE number is smaller than the referenced PE number. But the introduction of indirect exportation has made this binding order rule no longer sufficient, as shown in the following example. Suppose $i < j$. PE_i exports its unbound variable X to PE_j (resulting in an EXREF cell X') and PE_j indirectly X' back to PE_i (resulting in an EXREF cell X''). Since $i < j$, PE_i is allowed to bind the variable X to X'', creating a reference loop.

We have introduced the notion of *safe* and *unsafe* external references and modified the binding order rule to fix this problem.

Definition 1 *An external reference E is* unsafe, *iff*

(1) $processor(from(E)) < processor(to(E))$ [6], or

(2) $to(E)$ is an unsafe external reference.

An external reference E is safe *if it is not an unsafe reference.*

Since the second disjunct of unsafeness definition cannot be checked locally, an *unsafeness* flag is introduced, so that the criteria of unsafeness is as follows: An external reference E is unsafe iff (1) $processor(from(E)) < processor(to(E))$, or (2) the unsafeness flag of E is ON.

When a term is encoded by location, the unsafeness flag in the encoded form is set to ON if the term is an unsafe EXREF cell, OFF otherwise. When an EXREF cell is encoded by reference, the state of the unsafeness flag is inherited.

The binding order rule An exported unbound variable [7] cannot be bound to an unsafe EXREF cell. [8]

[6]The order is the reverse of that in [Ichiyoshi 87] and [Foster 88a]. The reason is to make the argument valid for a machine consisting of infinitely many processors.

[7]Such a variable has a tag EUNDEFin our implementation, separate from the UNDEF tag for non-exported variables.

[8]It does not apply to EXVAL cells, since they can never take part in reference loops.

To prove the reference loop avoidance, we add a couple of assumptions: (1) There is no reference loops at start-up time, and (2) It is guaranteed that reference loops made up of only internal references are not created.

We show below that reference loops will never be created by *reductio ad absurdum*.

Let L be a reference loop. By dereferencing internal references, we can safely assume that it consists of external references alone. There can be three cases:

case 1 L is made up of safe references alone.

case 2 L is made up of unsafe references alone.

case 3 L is made up of both safe and unsafe references.

Case 1 is impossible because every safe reference is from a PE with a larger number to one with a smaller number. Since exporting of cells alone does not make a reference loop, there must have been a binding of a variable to an external reference. By the binding order rule, case 2 is ruled out. If case 3 holds, there must exist a safe reference whose referenced data is an unsafe EX cell — a contradiction to the definition of a safe external reference and the binding order rule.

As a corollary, we show that dereferencing process — internal dereferencing by tracking REF chain and external dereferencing by passing *%unify* messages — always terminates. Suppose some dereferencing never terminates, it must be that the dereferenced result which is an unbound variable becomes bound to a reference (internal or external) to another unbound variable which in turn becomes bound, and so on, during dereferencing, so that the final dereferenced result will never be reached. Let $x_1, x_2, \ldots$ be such a "descending sequence" of cells, that is, every x_i is originally an unbound variable which is then bound to the reference to x_{i+1}. The sequence of PE numbers of the PEs in which those cells reside constitute a non-increasing sequence by the binding order rule. Any such sequence of natural numbers has a minimum element. After that minimum PE is reached, no external references appear. The problem is thus reduced to termination of internal dereferencing. This can always be guaranteed for a uniprocessor model (such as Multi-PSI), or can be guaranteed by introducing a local binding order rule for a shared memory multi-processor model (such as PIM).

6.2 Termination of unification

As long as the binding order rule is observed, we are free to choose any strategy as to when and to which PE unify messages are to be sent. Our strategy is (1) to do local unification whenever possible, (2) to shorten the sum of the lengths of the external reference chains (e.g. If the WEC of one of the references is 1, make it the first argument of the unify message. If it was made the second argument, indirect exportation would make the reference chain longer.), and (3) to send a unify message to the PE in which unification is likely to terminate (e.g. Send the unify message to PE with a larger PE number.). We omit the details in the paper.

We prove here that distributed unification between two non-circular terms terminates. Actually, we only show that every active unification is eventually reduced to local unification. That is, it does not keep on just passing *%unify* messages between PEs forever.

Suppose active unification between two terms X and Y is tried. Each of the dereferenced results of X and Y, namely DX and DY, is either an unbound variable or a concrete value. If we assume for simplicity that no binding occurs on DX or DY during the unification process, it can be shown by induction on the sum of the lengths of the external reference chains starting from X and Y that every unification is eventually reduced to local unification. Indirect exportation has to be taken special care because it makes the external reference chain longer. By making an external reference with WEC=1 the first argument in unify message whenever possible, this problem can be worked out. In general, the unbound variable DX (DY) which is the dereferenced result of X (Y) may become bound to some value during unification. But the number of such bindings can be only finite, as is shown in the argument for dereferencing termination. After the last binding is made, unification termination is guaranteed as in the simple case.

7 WEC ALLOCATION STRATEGY

As far as the WEC maintenance operations observe the WEC invariant, they are free to choose any values for WEC. But, if exhaustions of WEC at reference duplication happen very often because of a bad WEC allocation strategy, the performance is degraded. We give the simple strategy we employ in the KL1 implementation on the Multi-PSI.

The WEC of an export table entry is represented by a 64 bit unsigned integer, and the WEC of any external reference (import table entry and encoded reference) is represented by a 32 bit unsigned integer. We do not have to worry about overflow of the WEC of an export table entry, because it is impossible hardware-wise that there exist more than 2^{32} references to a single export table entry in the system (PIM or Multi-PSI) simultaneously. When the WEC of an import table entry overflows, 2^{24} is left and the excess is returned to the export table entry by a *%release* message.

The WEC to assign in encoding by location is always 2^{24} At reference duplication, the WEC of the import table is divided in half. It follows that an external reference which is encoded by location can be duplicated at least 24 times until the WEC becomes 1. Since relatively few data are exported and then duplicated more than 24 times, the rate of WEC exhaustion in all reference duplications is expected to be very low.

The WEC to assign in encoding to access is 1 when the WEC of the export table entry is greater than 1, and 0 otherwise (zero encoding), unless it is known by the MRB mechanism that the last reference to the EXcell is being consumed, in which case all WEC is attached.

8 RELATED WORKS AND DISCUSSION

The distributed unification in the parallel implementation of Flat Concurrent Prolog (FCP) [Taylor 87] involving variable migration is a very complicated procedure. This is because the unification in FCP at commitment has to be an atomic operation. Since unification is done locally, reference loop avoidance and termination of unification is easier to assure.

[Foster 88a] (or, [Foster 88b], a shorter version) gives a distributed unification algorithm similar to ours, though garbage collection is not addressed. The same binding order rule as in [Ichiyoshi 87] is used to avoid creation of reference loops. Whereas the symmetric protocol (*s_unify*) is used for termination detection in [Foster 88a], we devised a termination detection mechanism [Rokusawa 88] that does not require message acknowledgments.

Independently from us, Foster recently employed weighted reference counting garbage collection in his distributed implementation of the Strand language [Foster 88c]. Though his implementation does not have incremental local garbage collection mechanism, the single-assignment property of Strand allows release of weights at *Assign* message sending. Unlike our export and import tables, duplicate entries are allowed in the Incoming Reference Table (IRT) and the Outgoing Reference Table (ORT). In the KL1 implementation on the Multi-PSI, we try to minimize the overheads associated with the full WEC mechanism by restricting its use to those data which may have more than one reference to them; when exporting a data that is known by the MRB mechanism to have only one reference to it, we use a much simpler scheme.

In Foster's unification procedure, after one of the arguments has been fully dereferenced by *unify1* messages, the other argument is dereferenced by *unify* messages. Since this guarantees that an unbound variable is never bound to an indirectly exported variable, there is no need of introducing the safeness flag. However, unlike our unification procedure, a unify message must always be sent when unifying an unbound variable with an external reference, because the reference might be *unsafe* in our sense. [9]

One problem with the WEC scheme is that circular structures extending over PEs cannot be reclaimed. This is true with any reference counting garbage collection. Circular structures arise in AND-parallel languages when (1) the program explicitly creates circular data or, (2) two or more processes communicate with each other through shared variables (the goal records and the shared variables constitute the circular structure). A circular structure of the second kind gets untangled when the constituent processes terminate successfully, but remains as garbage if the processes are aborted or go into a deadlock state. We do not know yet how serious this problem of non-reclaimable garbage is. Eventually, we might need to implement global garbage collection.

The new external reference mechanism and the unification algorithm are adopted in the KL1 implementation on the Multi-PSI version 2 (instead of the old PSI CPUs used in the Multi-PSI version 1, it uses the CPUs of PSI-II machines [Nakashima 87]). The implementation is near completion and we will soon start running benchmarks for evaluation.

ACKNOWLEDGMENTS

We would like to thank the members of the KL1 implementation group in ICOT for stimulating discussions. We are also indebted to Dr. S. Uchida, the director of the 4th Research Laboratory, and Dr. K. Fuchi, the director of ICOT, for giving us the opportunity of research in this area.

REFERENCES

[Bevan 87] D. I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of Parallel Architectures and Languages Europe*, pages 176–187, June 1987.

[Chikayama 87] 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.

[Clark 86] K.L. Clark and S. Gregory. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1-49, 1986.

[Foster 88a] I. Foster. *Parlog as a System Programming Language*. PhD thesis, Imperial College of Science and Technology, March 1988. To be published by Prentice Hall.

[9] Actually, Strand semantics states that an attempt to bind a variable to a term containing that variable is an error. How this is to be enforced, however, is another question.

[Foster 88b] I. Foster. Parallel implementation of parlog. In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. II Software*, pages 9–16, 1988.

[Foster 88c] I. Foster. An asynchronous parallel garbage collector for a single-assignment language. To appear as Imperial College Technical Report, 1988.

[Goto 87] A. Goto. Parallel inference machine research in FGCS project. In *Proceedings of US-Japan AI Symposium 87*, pages 21–36, 1987.

[Goto 88] A. Goto, Y. Kimura, T. Nakagawa, and T. Chikayama. Lazy reference counting — an incremental garbage collection method for parallel inference machines. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1241–1256, 1988.

[Ichiyoshi 87] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of Flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 257–275, 1987.

[Nakajima 88] K. Nakajima. Piling GC — Efficient Garbage Collection for AI Languages. Technical Report TR-354, ICOT, 1988. Also to appear in Proceeding of the IFIP WG 10.3 Working Conference on Parallel Processing, 1988.

[Nakashima 87] H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine : PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, pages 104–113, September 1987.

[Rokusawa 88] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. I Architecture*, pages 18–22, 1988.

[Shapiro 84] E. Shapiro. Systolic programming: A paradigm of parallel programming. In *Proceedings of The International Conference on New Generation Computer Systems 1984*, pages 458–470, 1984.

[Shapiro 83] E. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, ICOT, January 1983.

[Taki 86] K. Taki. The parallel software research and development tool: Multi-PSI system. In *Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium 1986*, pages 365–381, 1986.

[Taylor 87] S. Taylor, S. Safra, and E. Shapiro. A parallel implementation of Flat Concurrent Prolog. *International Journal of Parallel Programming*, 15(3):245–275, 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.

[Watson 87] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proceedings of Parallel Architectures and Languages Europe*, pages 432–443, June 1987.

APPENDIX

A THE COMPARISON BETWEEN WRC AND WEC

We briefly compare our WEC scheme with the WRC scheme in [Bevan 87] and [Watson 87].

1. WEC has export and import table to reduce the number of inter-processor read requests. The export table also makes independent local garbage collection feasible.
2. The addition of WEC at importation does not have its counterpart in WRC.
3. The WEC supply protocol and zero encoding do not have their counterpart in WRC.
4. The notion of safe and unsafe external references is not needed in WRC, since WRC is not applied to logic programming languages.

Of course, all these extra features in WEC have overheads associated. In particular, log encoding optimization adopted in WRC is impossible in WEC, because WEC can be added at importation. The trade-off depends on the language as well as the ratio between intra- and inter-processor communication throughput.