A Non-shared Binding Scheme for Parallel Prolog Implementation

Kang Zhang **Ray Thomas**
Department of Electrical and Electronic Engineering
Brighton Polytechnic
Moulsecoomb, Brighton BN2 4GJ
UK

Abstract

To allow efficient parallel processing of Prolog programs on distributed multiprocessors, a non-shared variable binding approach is required such that binding environments can be independently distributed among processors. This paper presents a binding scheme, which realises the independence of a clause's binding environment by eagerly instantiating variables across clause arguments. The application of the scheme on a Prolog virtual machine has illustrated its features of efficiency in execution and simplicity in implementation. The preliminary performance evaluation has demonstrated the feasibility of the scheme.

1 Background

Prolog has been widely recognised for the past decade as a powerful symbolic programming language. To improve the execution speed of Prolog programs, much attention has been devoted to the parallel implementation of the language. One of the central issues is how variable bindings are represented and manipulated so that AND/OR parallelism can be supported most effectively.

The conventional approach to representing binding environments (BEs) in Prolog is known as the "three stack" approach [Werren, 1983]. Much of the work on parallel implementations of logic programs has been on efficient ways of implementing a parallel version of the "three stack" approach. The principle is that when a new process is spawned it can share the BE of its parent process, and make its own variable bindings along the new branch on the stack.

Various kinds of execution model have been investigated for the efficient management of variable bindings. Recent achievements in implementing these models on conventional von Neumann computers have been highly successful [Baron *et al.*, 1988; Ciepielewski, 1989; Lusk *et al.*, 1988]. The central idea in those parallel implementations is to build a virtual stack for each process so that it can share as much information as possible with its sibling processes. The binding schemes in these models have the following concepts in common:

a) variable bindings are kept locally in individual clauses;

b) unification of a goal and the head of an applicable clause often requires access to variables which may be bound earlier;

c) the unification between two unbound variables is realised by binding one variable to the reference of the other.

The differences between these schemes are that, to dereference an ancestor variable, different types of auxiliary structures are used (e.g. binding arrays or hash windows) to allow each clause to store its own copy of the variable [Conery, 1988].

Systems with centralised auxiliary structures may be called *shared binding schemes*. The major disadvantages of shared schemes are that, to access a variable which is bound at a very early stage, the dereference operation is sometimes costly; also the link of the BEs generated at different resolution stages requires a centralised memory organisation to facilitate the auxiliary structure. These result from the history sensitivity of variable bindings, in other words, any variable binding has to be kept in order for descendent processes to access.

In *non-shared binding schemes*, however, the number of BEs seen by any process is restricted to one or two. Such schemes overcome the above drawbacks but include a limitation due to tradeoff in extra copying and binding operations. Among the few proposed non-shared binding schemes, the EPILOG model [Wise, 1986] and the closed environments [Conery, 1988] are most representative. EPILOG allows only one BE to be accessed by a process, this is achieved by *back-unification*. In closed environments, locality is achieved by a two-stage closing operation on both parent and child environments. Both of these approaches represent a radical attempt at achieving locality for a non-shared implementation. But back-unification in the former and closing operations in the latter are considered costly and sometimes wasteful in both computing power and use of storage, because variables are identified locally in individual processes in both systems.

In the following section, a new scheme is presented that identifies variables in a global space so that back-unification is reduced to simple variable exportation, and also reduces environment space to a minimum by eagerly instantiating variables in local processes.

2. Eager Instantiation Binding Scheme

To design a binding scheme suited to a scalable multiprocessor, high distributability of BEs is desirable. In other words, BEs should be distributed, rather than shared among processes. Operations associated with shared stacks and registers should, therefore, be avoided.

It is generally true that high distributability is usually achieved at the cost of structure copying. If the amount of data to be copied is as large as the whole auxiliary structure for BEs, the copying overhead may well outweigh the performance gain of distributing the processing. Our first goal is to minimise the size of the BEs to be copied. Secondly, the binding scheme should be as readily supportive as possible for both OR and AND-parallelism. Thirdly, it should be relatively inexpensive for the system to allow both parallel execution and sequential execution and to switch from one mode to another.

The eager instantiation binding scheme is designed to meet the above requirements. A major feature of the scheme is that the number of BEs operated on by a process at any instant in time is restricted to one, this is the process' own BE.

It is observed that, the evaluation of a process can be entirely independent of the precedent (sibling or parent) processes if the variables which appear in the input arguments of the process, and are bound during the previous unification, have been substituted by their instances according to the previous BE. In other words, as long as bindings created in the precedent process are applied to the corresponding variables before the variables are sent to the current process, any access to the variables needs not be dereferenced to the ancestor BEs.

In this binding scheme, the above condition is guaranteed by variable instantiation operations, which ensure that any bound variables appearing in an argument will be substituted with their corresponding instances in the BE. When structured terms contain variables for substitution, they are reproduced with the variables being substituted by their instances. Instantiation operations have to be applied to the process arguments which are due to be transferred to the next process.

Argument transfer may happen either a) between sibling processes, or b) from a parent process to its child process after head unification. In the first case, the instantiation of bound variables has to be made before the results of the process are sent to the output, by using the process' own BE, and before the input to a sibling process, by using the binding results of the previous sibling process. This is called *interface instantiation*. In the second case, the instantiation has to be done after the head unification and before the evaluation of the child processes for the body literals. This is called *face instantiation*. Once the evaluation of a child process terminates, the ancestor variables that were bound in the child process are paired with their binding instances and exported back into the parent process. This export operation, together with interface instantiation, achieves the same goal as the back unification operation in EPILOG [Wise, 1986]. They ensure that the binding results of a process will not be required later by its parent or sibling processes. The face instantiation, on the other hand, serves to ensure that the child processes have sufficient binding information from their parent so that they need not to dereference back to it.

Figure 1 illustrates the eager instantiation binding scheme applied to a clause. Within the clause, process A has two goals B and C. After head unification of A, argument X is instantiated (face instantiation), and then transferred to goal B. When the evaluation of a child process which matches B succeeds with X bound to a value v, then the variable-instance pair X/v is sent back to A to be used for the instantiation of Y (shown as an arrow in Figure 1). It should be noted that more than one variable-instance pair may be produced, if X, introduced from the head, is a structured term containing more than one unbound variable. Argument Y (which may be bound to a structured term f(X)) is instantiated (to f(v) - interface instantiation) before being sent to goal C. Upon the successful evaluation of A, all arguments are instantiated according to A's updated BE (interface instantiation) and then sent back to A's parent process. X and Y in the example merely stand for clause arguments - they may be any terms.

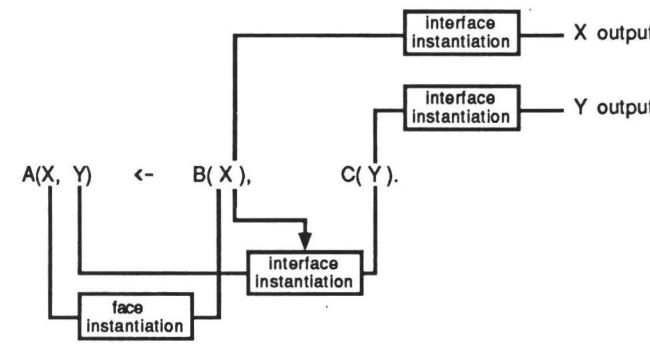

Figure 1 Eager instantiation scheme

When more than one child process matches goal B the multiple binding instances of X, resulting from the unification of these processes with B, will be stored in a special data structure, called *stream*, or v_s. The variable-instance pair X/v_s will be used for the instantiation of Y, which generates a stream of Y's instances.

From the above discussion, it is observed that full AND-parallelism is not allowed in principle with the scheme. But the scheme can effectively exploit Restricted AND-parallelism [DeGroot, 1984], which is parallel processing of run-time independent goals. There should be no interface instantiation between the independent goals because exporting variable-instance pairs from a child process is necessary only when the corresponding goal in the parent process has run-time data dependencies with one or more sibling goals. In other words, the child processes that are independent from each other at run-time can be evaluated in parallel without exporting their variable-instance pairs to the parent process. Therefore, the arrowed line in Figure 1 can be omitted in this case.

The binding scheme is called eager instantiation due to the fact that the instantiation operation is eagerly performed by the local process. Once a variable has been bound, all subsequent occurrences of the variable will be replaced by its binding instance. By ensuring this, the scheme introduces more overheads on instantiation operations and structure copying, but it offers a simpler and more distributed parallel implementation by eliminating the need for accessing variables in the ancestor process.

With eager instantiation of procedure arguments, the evaluation of a logic program can be best interpreted in terms of procedural semantics [Hogger, 1984]. Every reduction step in a computation entails invoking some

procedure (parent process) and thereby introduces new calls to the goals (child processes) in the procedure's body. The process of replacing a call by a goal in the body involves only making the call arguments and goal arguments correspond. This is very similar to the way in which the semantics of procedure calling is defined for conventional languages. The only difference is that logic programs make two arguments correspond by unifying them, whereas conventional languages impose different (simpler) semantics. The unification of a procedure may produce a BE, which only effects the calls to the body goals.

In representing a variable, the concept of a "value cell" [Gabriel *et al.*, 1985] is not used. Instead, a variable-instance pair is used to accommodate a variable and its binding, and is created dynamically only when the variable is bound. This arrangement, though, loses the advantages of a value cell, i.e. a variable instance can be directly accessed because its address is the variable itself and binding conflicts can be checked efficiently, but it brings the following benefits.

Variables are represented uniquely by variable symbols, which are generated at run-time as the evaluation proceeds. It is unnecessary to provide memory space for unbound variables (whereas in a value cell implementation any variable, bound or unbound, occupies a physical space). Therefore, there is no synchronisation problem associated with simultaneous writing to (binding) a variable by different processes. A variable-instance pair occupies two spaces for X/B, which is created only when the variable is bound to another term during unification. (In an OR-parallel implementation using value cells, a value cell may also need two or more spaces). Furthermore, a variable can be presented in a number of pairs if it is bound to different values as in OR-parallel execution.

Generally, the BE size of a particular process can be minimised by optimisation. This is done by substituting variables in the instance parts with their own instances found in the same BE, and removing the redundant variable-instance pairs which are no longer needed. The optimised BE may occupy much smaller space than the space of value cells required for all variables. Thus BEs may be extensively copied between processors on requests in order to exploit more distributed processing capability.

3 Application on a Dataflow Virtual Machine

The eager instantiation binding scheme was initially designed for a dataflow Prolog execution model, called DIALOG [Zhang, 1990]. The DIALOG model is built on a virtual machine and based on dataflow computation.

Dataflow computation allows the operations which have received all their input data to be executed in parallel. The computational model is typically restated as two-dimentional graphs, known as *dataflow graphs*, which show the data dependencies among operations.

The DIALOG virtual machine consists of a number of virtual instructions, of which dataflow graphs representing individual Prolog clauses are constructed. The instruction nodes in a dataflow graph are connected by arcs along which dataflow tokens are transferred. All the nodes whose input arcs have received tokens can be executed simultaneously. Each node, after execution, may put a new token representing a result on its output arcs, and thus activate the following nodes.

To see how the eager instantiation binding scheme is applied, consider a simple example which shows one situation where variable instantiation is required:

```
<- choose_list(X1, [X1|X2]).
choose_list(X, Y) <- element(X), list(Y).
                     element(a).
                     list([a, b, c, d]).
                     list([b, c, d, e]).
```

The dataflow graphs for individual clauses are shown in Figure 2. A brief description of the function of instruction nodes in the graphs follows. More detailed descriptions on the instructions and some rules for generating dataflow graphs are given in [Zhang, 1990].

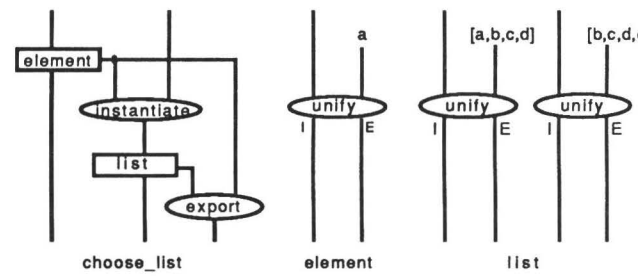

Figure 2 Dataflow graphs for choose_list program

The instructions represented in rectangular nodes perform procedure calls. A procedure call instruction transfers the argument values from the process named in the node to the clauses whose clause heads match the process, and thus activate the clauses. Meanwhile, the instruction sets the returning pointers in the clauses pointing to the descendant processes in the parent process.

Activated by an input term and a BE, an *instantiate* node substitutes all the variables appearing in the term with their binding instances recorded in the input BE, and sends the instantiated term to its output. An *export* node merges the input variable-instance pairs where the variables were created by the ancestor processes, and exports the new BE to the parent process.

A *unify* instruction node unifies two input terms and generates an instance output I and an environment output E. The I output produces a common instance of the two terms. The E output delivers a BE which records bindings of the variables appearing in the two terms.

The execution sequence of the above example is illustrated in Figure 3.

The query implies that the first element of the chosen list must be found in element. When the query is executed, X1 and [X1|X2] are applied to the graph of choose_list (Figure 3(a)). According to the DIALOG virtual machine, head unification needs not to be applied to any stand alone variables in the head. Therefore, in this example, X1 is sent straight to the first child process element (Figure 3(b)). After unification of element, the BE with one variable-instance pair {X1/a} is produced and exported to the parent process choose_list (Figure 3(c)). The second argument input is instantiated to [a|X2] before being sent to the

second goal list (Figure 3(d)). The child process list, activated by the instantiated argument [a|X2], succeeds with the first clause and fails with the second (Figure 3(e)). Finally, the instantiated outputs a and [a, b, c, d] are returned to the corresponding two arguments of choose_list (Figure 3(f)). Since variable X1 and X2 were imported from a parent process (the query in this case), their bindings X1/a and X2/[b,c,d] are exported through an *export* instruction, from element and list respectively, back to choose_list's parent process. It is clear that the evaluation of the child process list need not access the BE of element, nor the BE of choose_list.

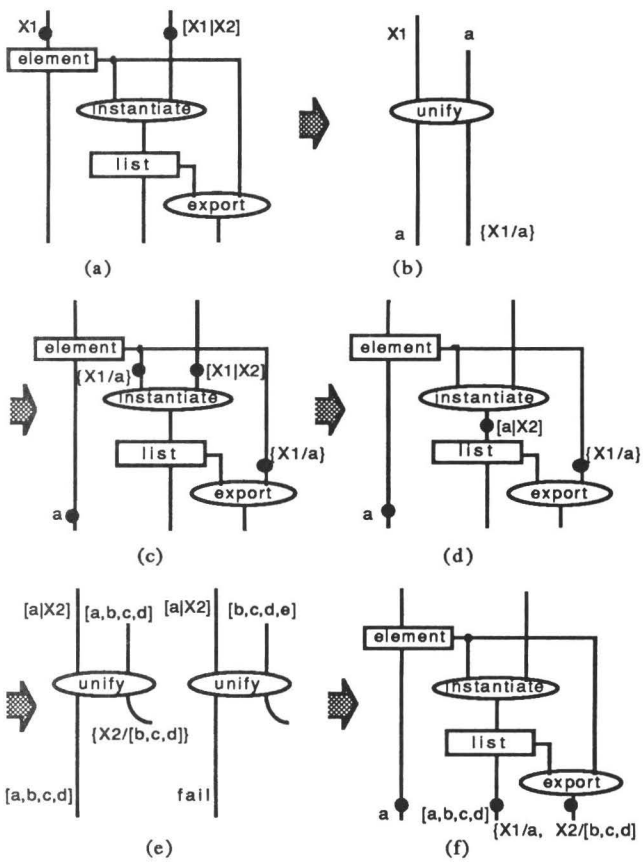

Figure 3 Snapshots of applying the scheme to choose_list

4 Implementation and Performance

The DIALOG model has been implemented in Occam2 on a transputer system [Zhang, 1988].

4.1 Data Representation

Variables are identified by their variable symbols, and created when they have been bound. A BE is represented as two arrays of integers, one storing variables, the other storing binding instances. Two arrays together form a whole array of variable-instance pairs. When two unbound variables are unified, the variable-instance pair is represented thus: the more recently created variable is the binding instance of the other.

Constants are identified by pointers pointing to their values, which may be integers or character strings.

Lists are identified by list pointers pointing to the starting addresses of the individual lists represented as:

arity (N)	No. of unbound variables	element_1	...	element_N	tail

Structures are identified by structure pointers pointing to the starting addresses of the individual structures, represented as:

functor	arity (N)	No. of unbound variables	element_1	...	element_N

Lists and structures are referred to as *structured terms*. The arity of a structured term is the total number of elements in the term. The total number of unbound variables is another important status of the structured term. It helps by eliminating unnecessary searches for variables in a non-variable structured term, and also plays a key role in the implementation of Restricted AND-parallelism.

4.2 Garbage Collection

The structure copying policy causes the memory space to be consumed rapidly. Therefore, garbage collection is frequently needed to reclaim memory space occupied by structured terms no longer used.

The simple reference count garbage collection scheme is found to be sufficient to solve the problem. Each structured term in the structure store has an extra tag indicating its reference count. When a structured term is first created, its reference count tag is assigned to be one. When the reference to the term needs to be copied (e.g. when a goal including the term is unified with a number of applicable clauses), the tag is incremented by one for each copy. Conversely, whenever a reference to it is discarded, the reference count decrements by one. For example, when the term is updated by instantiating variables in it using the existing binding information, the updated term occupies a new space, and thus identifies itself as a different term. In this case, the reference to the old term can be discarded and replaced by a new reference. Once the reference count of a structured term reaches zero, the memory space for that term can be reclaimed as the term is no longer used.

4.3 An Abstract Machine Architecture

In order to study the feasibility and run-time characteristics of the scheme, an abstract dataflow architecture has been simulated. It should be noted, however, that the binding scheme is equally suited to the implementation on many other kinds of architectures. The dataflow architecture is constructed as a ring connected by the following main components (Figure 4): an instruction store where compiled virtual instructions are stored; a packet queue based on a FIFO for buffering packet flow; a number of homogeneous processing elements (PEs); a structure store for storing structured terms; and a distribution network for delivering result packets to the instruction store.

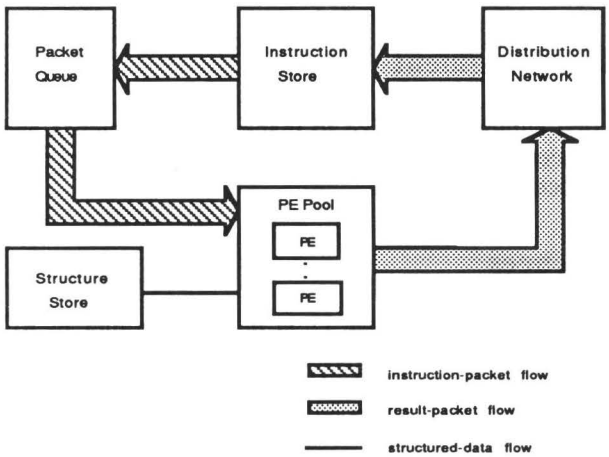

Figure 4 An abstract dataflow architecture

In this dataflow architecture, the information elements, called *packets*, are flowing simultaneously in different parts of the ring on behalf of different instructions for concurrent execution. Hence the ring operates as a pipeline with all of its components actively processing or creating packets simultaneously.

When an instruction fails in its execution, the responsible PE immediately sends a "fail" signal to the instruction store so that no more instructions belonging to the same graph are allowed for execution. Only when the graph is activated again may the "fail" signal be disabled.

4.4 Performance

Six programs have been tested to evaluate the effect of the binding scheme. They have been hand-compiled into dataflow graphs to be stored in the instruction store. The programs are

1) Determinate list concatenation for a list of 8 elements (append1);
2) Non-determinate list concatenation of a list of 8 elements with all results (append2);
3) Quicksorting program of a list of 30 elements in a reversed order (qsort);
4) Naive reverse of a list of 30 elements (na_rev);
5) List checking of a list of 4 elements as a subset of a list of 8 elements (sublist);
6) Line presentation program for a cube (cube).

Their evaluation reflects different characteristics: non-determinism in "append2" and "sublist"; recursion in "append1", "append2", "qsort" and "na_rev"; graph copying in "sublist"; and stream processing in "cube". However, the programs do not cover the whole range of benchmarking, though they have closely met the requirements of this phase of evaluation.

The following is a description of the performance of the scheme, measured in terms of BEs and structured data. In Table 1, the maximum lengths (number of variable-instance pairs) of BEs (EnvLen_Max) are less than three and the average BE lengths (EnvLen_Ave, averaged over all accessing instants by PEs) are less than two. This results from eager variable instantiation and the subsequent BE optimisation that merges variable-instance pairs in place.

m'suremt	PEs	append1	append2	qsort	na_rev	sublist	cube
EnvCpy_Tot [%]	1	0 [0]	0 [0]	0 [0]	0 [0]	0 [0]	0 [0]
	2	14 [70]	28 [53]	1000 [66]	677 [70]	26 [50]	0 [0]
	4	18 [90]	41 [77]	1145 [75]	929 [96]	37 [71]	0 [0]
	8	18 [90]	50 [94]	1491 [98]	951 [98]	42 [81]	0 [0]
	16	18 [90]	51 [96]	1515 [99]	956 [99]	44 [85]	0 [0]
EnvLen_Max		1	2	2	1	3	0
EnvLen_Ave		1	1.7	1.3	1	2	0

Table 1 Measurements of BE operations

A BE stays in a PE after its processing. When another PE needs to access the same BE next time, it will find, from the instruction store, the host PE where the BE resides, and then copy it across. Meanwhile, the PE sends its own identifier to the instruction store, thus associating itself with the copied BE. After being processed, the BE will stay in the second PE until it is required again. This lazy copying policy works on a probability basis because there is a probability that the BE is required by the same PE for two or more successive updating operations, in which case copying is unnecessary. When only one PE is used in the system, the total number of BEs copied by PEs (EnvCpy_Tot) is zero, so is the copying percentage [%]. The program "cube" involves no BE operations, and thus no copying is needed. The copying becomes dominant (mostly over 90%, except "sublist" 81%) when more (than 4) PEs are involved. Since the average BE sizes are almost minimum, the copying overhead is reasonably affordable.

The eager instantiation binding scheme requires extensive copying of structured terms, which is a major shortcoming of the scheme. Table 2 summarises the measurements on structured terms. The first row is the average number of structure elements involved in each program (StrLen_Ave), averaged over all accessing instants by PEs. The second row is the total number of accesses to the structured terms (StrAcc_Tot). It is found that the number of copying operations (StrCpy_Tot) is proportional to the number of structure elements being processed (EleCpy_Tot(1)), and to the number of logic inferences in the programs. On average, there is 1.3 copying operation per logic inference. Based on the evaluation results of the overall execution speed of the programs [Zhang, 1990], the copying policy does not impose intolerable overhead ("na_rev" program, which copies 17.6 structured terms per logic inference on average, performs even better than others). The copying problem may, however, become more serious when longer structured terms are involved.

measurement	append1	append2	qsort	na_rev	sublist	cube
StrLen_Ave	7.2	7.3	12.1	13.1	7.1	0
StrAcc_Tot	50	64	4499	2757	190	0
StrCpy_Tot	24	31	1770	1334	87	0
EleCpy_Tot(1)	173	226	21417	17475	621	0
EleCpy_Tot(2)	99	100	9400	8062	336	0

Table 2 Measurements of structured terms

One way to reduce the amount of copying is to allow some elements to be shared among different structured terms. Knowing that operations on structured terms in logic programs are overwhelmingly performed on list heads and tails, an intuitive idea is to separate heads and tails from other parts of lists. Therefore, when a new list is created whose head or tail has been modified from an old one, it may share the remaining part of the old list. This is illustrated in Figure 5.

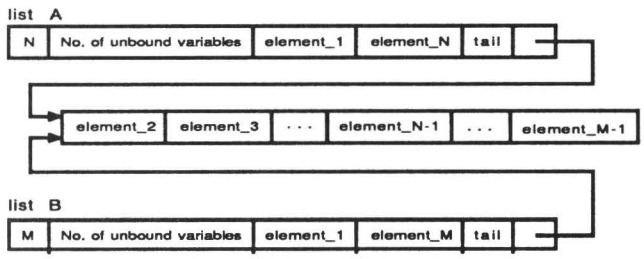

Figure 5 An improved list representation

The situation in Figure 5 happens when the list tail of list A is bound to a new list, whose elements correspond to element_N, element_N+1, ..., element_M-1, element_tailB. This is one of the most common situations in list processing. As Figure 5 suggests, the elements in list A, element_2, element_3, ..., element_N-1 need not be copied. The newly constructed list B has its own version of arity, number of variables, first element, last element and tail element. It has the same pointer as in list A pointing to the remaining elements of the list. The list arities (i.e. N and M) here tell the different belongings of the two lists. The amounts of copying of structure elements when the improved list representation is used appears in EleCpy_Tot(2), which is only about half (44 - 57%) of the amounts of copying using the non-shared representation (EleCpy_Tot(1)). Compared with the non-shared one, the shared representation introduces little extra overhead, that is incurred from tracing and copying a pointer.

5 Conclusion

A binding scheme called eager instantiation has been presented, which allows a variable to be bound and accessed locally in an individual binding environment so that no centralised auxiliary structure is required. Therefore, OR-parallelism can be exploited effectively within a distributed processing environment. It is also easy to be extended for support of Restricted AND- parallelism. The application of the scheme on a dataflow virtual machine has also been described. The scheme has low implementational complexity and high distributability.

The DIALOG execution model utilising the scheme has been evaluated and reported elsewhere [Zhang, 1990]. The performance obtained is encouraging on a simulated dataflow architecture with up to eight processors. It should be said, however, that the binding scheme is not architecture specific and is suitable for many other kinds of architectures. More complex programs that provide larger search spaces need to be tested to gain broader assessment of the scheme.

Acknowledgment

The first author would like to thank the UK Science and Engineering Research Council for the support of a postdoctoral fellowship.

References

[Baron *et al.*, 1988] Baron, U. *et al.* The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proc. Int'l. Conf. of Fifth Generation Computer Systems*, pages 841-850, Tokyo, 1988.

[Ciepielewski *et al.*, 1989] Ciepielewski, A., Haridi, S. and Hausman, B. OR-Parallel Prolog on Shared Memory Multiprocessors. *J. Logic programming*, 7:125-147, 1989.

[Conery] Conery, J.S. Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. *Int'l. J of Parallel Programming*, 17(2):125-152, 1988.

[DeGroot, 1984] DeGroot, D. Restricted AND Parallelism. In *Proc. Int'l. Conf. Fifth Generation Computer Systems*, pages 471-478, 1984.

[Gabriel *et al.*, 1985] Gabriel, J., Lindholm, T., Lusk, E.L. and Overbeek, R.A. A Tutorial on the Warren Abstract Machine for Computational Logic. Technical Report, ANL-84-84, Argonne National Laboratory, Argonne, USA, Jun. 1985.

[Hogger, 1984] Hogger, C.J. *Introduction to Logic Programming*. Academic Press, 1984.

[Lusk *et al.*, 1988] Lusk, E., Warren, D.H.D., Haridi, S. et al. The Aurora OR-Parallel Prolog System. In *Proc. Int'l. Conf. of Fifth Generation Computer Systems*, pages 819-830, Tokyo, 28 Nov. - 2 Dec. 1988.

[Warren, 1983] Warren, D.H.D. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[Wise, 1986] Wise, M.J. *Prolog Multiprocessors*. Prentice-Hall, 1986.

[Zhang, 1988] Zhang, K. An Occam2 Implementation of Prolog and Its Preliminary Performance. In *Proc. 9th Occam User Group Technical Meeting* , Askew, C. (ed.), Southampton, UK, Sep. 1988.

[Zhang, 1990] Zhang, K. DIALOG: A Dataflow Interpretation Approach to Logic Programs. Ph.D Thesis, Brighton Polytechnic, 1990.